Playing with sounds: a spatial solution for computer sound synthesis

Chapman, David P.

Award date:
1996

Awarding institution:
University of Bath

Playing with Sounds: A Spatial Solution for Computer Sound Synthesis

submitted by

David P. Chapman

for the degree of Ph.D

of the

University of Bath

1996

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

Summary

The fundamental problem of digital sound synthesis arises from the complete generality of the digital representation of sound. The large quantity of data required to achieve this generality renders its manual specification impossible. This has lead to researchers creating synthesis techniques that can generate sound given a smaller set of parameters. This, however, leads to a loss of generality.

The task of sound synthesis has, therefore, been split into the problems of instrument specification and score specification. Many different synthesis techniques exist which when coupled with the unit generator model provide an adequate solution for instrument specification. For any particular score representation system there will exist musical ideas for which it is too general, or there will exist ideas for which it is not general enough. A solution to this problem is to use the most appropriate notation for each idea that is a component of a work, and combine the results to produce the desired piece. Under the paradigm of experimental computer music this problem is shown to be one of supplying a technique for constructing composite sounds, from less complex components, which will allow for the modification of, and hence experimentation with, the composite without affecting the properties of the component sounds from which the composite is built.

This thesis describes a new model which provides a solution to the above problem by viewing the construction and modification of sounds as taking place in what are called parameter-spaces. Transformations of sounds are achieved via motion in space and not by altering the internal properties of sound objects. This thesis further describes a system which implements and tests this model. The results produced, and the background knowledge, are used to conclude that a computer music system for general sound synthesis with a completely controllable level of generality is possible.

Dedication

For

My grandfather,
William Henry Chapman

Acknowledgements

I would like to thank my supervisor Professor John Fitch for his advice on the research and the PhD process, and for getting me interested in computer music whilst I was an undergraduate. Thanks are also due to all the members of the Computing Group at the University of Bath, but in particular Simon Scott and Jetender Kang for managing to live with me for a number of years, and for many interesting discussions on philosophy, and of course football. To Richard Taylor and Bill Martin thanks are due for their considerable abilities at snooker, taking apart quiz machines, and consuming large quantities of beer, which have provided the basis for many entertaining evenings.

Uncountable thanks are due to the members of my family, and in particular my parents, John and June Chapman, for understanding that I function best when not put under any external pressure to excel, and for all the support and encouragement they have given me throughout my life. Thanks are also due to the EPSRC who have provided the funds, in the form of a grant, without which I would not have been able to undertake the research described in this thesis.

Finally, I would like to publicly extend many belated thanks to Mr André Previn, Mr Ernie Wise, and the late and much missed, Mr Eric Morcambe, for unknowingly inspiring me, many years ago, to take an active interest in music and its production.

Preface

This thesis is divided into three parts. The first part, which contains chapters one and two, takes the form of a literature review. The first chapter gives a very general introduction to the subject of computer music, and as part of this introduces the area of computer, or digital, sound synthesis. The second chapter reviews sound synthesis techniques and systems that have been designed in the past. From this critique deficiencies are cited in the existing systems, the solutions to which are described in part two.

Part two of this thesis, which contains chapters three and four, describes the theory, design, and implementation of a prototypical system which supplies a solution to the problems outlined in part one. Chapter three describes a new model upon which the solution and the prototypical system is based, and thus is to be considered as the crux of the work described in the thesis. Chapter four is concerned with the implementation and testing of the prototypical system which demonstrates the adequacy of the new model as a solution to the problems given.

Part three of this thesis summarises and evaluates the work described in the preceding parts. The evaluation consists of a summary, a description of possible future research, and the conclusions.

"It may be doubted whether human ingenuity can construct an enigma of the kind which human ingenuity may not, by proper application, resolve."

Edgar Allan Poe, The Gold Bug (1843)

Contents

List of Figures

Part I

An Introduction to Computer Music

Chapter 1

Computers and Music

1.1 Introduction

1.1.1 The Technology of Music

Humans have been composing and performing music for thousands of years. Whistles and flutes made from perforated phalange bones have been found at upper Palaeolithic sites of the aurignacian period (c. 25,000 - 22,000 BC) e.g. at Istallosko, Hungary and Molodova, formerly USSR [34]. This implies that the development of musical instruments has been ongoing for at least the last 24,000 years. While the oldest surviving notated composition dates much later, c. 1800 BC[1] [7], it is reasonable to suggest that composition must have begun around the same time as the production of these primitive instruments.

It is likely that these primitive instruments were limited in terms of the number of sounds that could be achieved with their use. The style of the music produced on these instruments would therefore be influenced by these limitations. This is a well established pattern in the evolution of instruments and the music they are used to perform. The general technology of a certain period in history will obviously have a bearing on the technology of instrument design and manufacture. Hence the technological advancement over time will affect the nature of musical instruments and thus musical style. An example of this process is the development of the modern trumpet.

The classical trumpet, as known to composers such as Haydn, Mozart, and Beethoven, consisted of a single brass tube folded twice, with a bell and mouthpiece at opposite ends [47]. This allowed the production of pitches resulting from an harmonic series beginning at the pedal tone of the instrument. However, the unpleasant quality of the lowest partials, and the difficulty in playing a steady tone pitched above the twelfth

[1] An Hurrian love song.

partial, restricted the range of the instrument to a set of ten different notes from a single tube length. The tube length could be changed, however, via the addition of a crook. The result of which was to enable the instrument to be played at the pitches arising from the harmonic series beginning at the new pedal tone. However, the addition or changing of the crook, took a sufficient quantity of time to render the operation impossible during a performance.

The addition of valves to the trumpet, in the middle of the nineteenth century, allowed for the effective length of the tube to be changed, almost instantaneously, with their depression. Hence the valve trumpet was capable of producing notes from a chromatic scale between its third and twelfth partials. This increase in range of sounds lead to composers writing parts for the trumpet of greater complexity and agility. The trend towards greater complexity lead in the twentieth century to the adoption of the smaller, but more agile, modern trumpet. This latest development has allowed for the evolution of many different styles of trumpet playing. Hence the modern trumpet is as equally at home in a symphony orchestra, as in a jazz or popular band. As such it has had an influence on the development of many musical genres.

The development of most instruments follows this same pattern. Firstly, the current instrument is considered restrictive. New technology then allows for the production of a new instrument, based upon the older one, which is less restrictive. This new found freedom allows for new effects, often not considered when the instrument was conceived, which expands the range of musical expression. Given this pattern, inherent in music history, it was almost inevitable that the technology of the electronic age would affect the nature of musical style and production.

1.1.2 The Origins of Computer Music

The idea of using a computer for musical purposes is almost as old as the computer itself. The work of Charles Babbage, in the 1840's, on his difference and analytical engines, both mechanical computers, prompted his friend and fellow mathematical Lady Ada Lovelace to suggest the possibility of such a machine operating on musical data using established rules of composition. In this way the machine could be used to generate a musical work [29].

Despite the invention of a number of electronic instruments such as the *Theremin*, named after its Russian inventor Leo Theremin, in the 1920's, it was not until the late 1940's and early 1950's that a computer was used to generate music. Unlike these early electronic instruments, these computer-based experiments did not actually produce sound directly. Instead the computer was used to generate musical scores, or rather the data required for their construction, with the aid of stochastics. One such

experiment, in 1955 by Lejaren A. Hiller Jr and Leonard M. Isaacson at the University of Illinois, resulted in the production of a work entitled the *Illiac Suite* for string quartet. Its production entailed the random generation of notes, by the computer, which were rejected if they violated some predefined rules. This continued until simple melodies were produced, which were then transcribed by hand into a performable score.

1.1.3 Contemporary Computer Music

From its rather humble beginnings in the late 1940's and early 1950's the subject of computer music has expanded into number of identifiable areas. These areas may be classified as systems for input/output storage of musical data, computer-based automated composition, performance systems, and sound analysis & synthesis. Whilst it is unlikely that all work in the past and present will fall neatly into just one of these areas, we will briefly examine each of them separately.

1. Systems for input/output storage of musical data

   An obvious use of the computer for musical purposes is in the typesetting of musical scores, for the purpose of publication, performance, and archiving. Just as text has the word processor, many musical score editor programs have arisen over the years, to meet the afore mentioned need. Until recently, however, the role of music notation in composition has been largely overlooked [8]. This is concerned with the sketches, in a particular notation, that a composer makes whilst composing a work. It is useful therefore to have not only a system which implements a set number of notation schemes, but also allows the design of new forms of notation, which are more appropriate to a particular work, which can be mapped, after completion, into a standard form. This we will remark upon at a later stage in the text.

   The production of music notation on computer maps a computer representation, manipulated graphically by the user of a system, ultimately onto a hard copy score. As a means of feeding musical data quickly and efficiently into a computer program, and hence into a computer representation of this data, work is in progress that seeks to produce a program that implements the reverse of the process of notation on computer. This is called optical music recognition (OMR) [13, 3], since it is an attempt to read printed or handwritten scores [65]. This forms a very useful mechanism for input to other types of music systems, such as those for composition, and also provides a mechanised method for archiving sheet music on a computer based medium, such as disk or magnetic tape.

2. Computer-Based Automated (Algorithmic) Composition

   In this area the aim is to produce computer systems that automatically compose music based upon some abstract model of the process of composition. This, as has already been noted, was the first aspect of music in which a computer was used. Much of the early work in this area employed stochastic, or random, processes and techniques, such as Markov processes, to generate notes. The work of the French composer Iannis Xenakis [67, 68] is of particular note in this field.

   In recent years, researchers have been examining the musical possibilities of chaotic [10], and non-linear [28, 27] systems. As with the use of stochastic processes, the hope is that these models will produce what we might call "good" music since, like such music, they are ordered but not in an obvious or repetitive fashion, that is they are in a sense simple and yet complex [2].

   More peripheral work in this field has included the use of logic programming [62], viewing a process such as harmonization as essentially one of problem solving under a given set of constraints. Further attempts at automated composition have focussed on such disparate areas as neural networks [14], and genetic algorithms [20, 23, 19]. The wide range of scientific models on which the work of algorithm composition is based has, not surprisingly, lead to the accusation that researchers are blindly jumping onto the current scientific bandwagon, and that the we should strongly consider the value of the results [55]. However, the musical value of the resulting work does not depend upon the correctness of the philosophy behind the use of a particular technique. Both stochastic, and non-linear (chaotic) approaches, were without doubt used, prior to an understanding of the relationship between these processes and what we have called "good" music. This relationship is now, however, beginning to be understood. Whilst this is not important from a practical point of view, it does put the subject of algorithmic composition on a more solid philosophical platform.

3. Performance systems

   Performance systems, as the title might suggest, are concerned with the control of sound, and instruments during the live performance of a musical work. The prime influence in this field is the language MIDI[2] [1], which was designed as a means of communication from, and hence providing control with, a standard piano-like keyboard or synthesizer, to other instruments. The principle mechanism employed in the use of MIDI is therefore that of data-flow control, and data

[2]MIDI is an acronym for Musical Instrument Digital Interface

modification, taking place between the line of communication from one MIDI interpreter and generator, say a keyboard, to another, possibly a sound synthesis program. The MAX [58] graphical programming language provides a tool for specifying, and controlling, the data-flow between the components (instruments, programs etc) involved in a particular performance. This is achieved via the linking of what are called *patches*, programmed by the user, which provide the functionality of data-flow control, and data modification. The ultimate output of a MAX program is MIDI data[3] used to control the components which form the orchestra for a performance.

MIDI has achieved much in the field of live performance, allowing many kinds of interactive performance such as automated jazz accompaniment [16], but has had something of a detrimental effect on the field of sound synthesis, an area which we will discuss later. By setting itself up as a note based standard for musical interchange, it has narrowed the focus of compositional thought into the production of streams of notes, rather than more general high level structures. As Max Matthews stated in 1974

> Direct digital synthesis makes it possible to compose directly with sound, rather than having to assemble notes.

The MIDI language, which has no inherent mechanisms for the control of timbres[4] [52], works against this promise, if not intentionally. The role of MIDI should be that of one of many input and output mechanisms to and from sound synthesis software, and not that of a protocol for all musical communication. A role for which it is far too restrictive. To quote Andy Moorer's aphorism regarding MIDI

> No adjustment necessary - in fact, no adjustment possible.

4. Sound analysis & synthesis

The analysis of the properties of sound can be seen to be of interest for purely philosophical reasons, but is for the most part used as a generator of models for sound synthesis, to which it is largely subservient. For example the production of the spectra of a sound [59] is useful for building a parameter driven (frequency and amplitude, say) pressure function having the same timbre as the sound in

[3]The Max/FTS program, distributed by IRCAM, also allows the input and output of audio data.

[4]General MIDI allows the indexing of a predefined, and finite, set of timbres. For example timbre 00 is grand piano. The ZIPI music parameter description language [33, 66] overcomes many of the problems with MIDI by providing a faster ring, package driven, architecture with a greater number of control parameters including an increased set of possible notes, and a model of timbre as a 3-space with dimensions of brightness, roughness, and attack character.

question. An analysis method such as source separation [35], in which a sound is broken down into its source components, is useful as an input to a synthesis program, specifying a number of note sequences for example.

Computer sound synthesis is concerned with the generation of data which represents sound, and in particular music. This is the subject on which this thesis is written. In the following text we will show that the problems in computer sound synthesis stem from the nature of sound, and hence the digital representation of sound used by computers. We will then proceed to describe how the problems, that remain unsolved, can be dealt with.

1.2 Digital Sound Synthesis

1.2.1 The Nature of Sound

The perception of a sound generated by some source in our world is the result of a particular physical action. When an object vibrates, i.e. moves about a given point, in a given medium, say air, it displaces the molecules which form the medium. These displaced molecules displace neighbouring molecules, and thus a wave propagates from our source object [60]. We say that the wave *travels* from the source by the means of these displacements in the medium.

Let us now suppose that we place a human observer within range of the propagated wave. The wave will reach the head of the observer, enter the ear, and travel down the auditory canal before finally striking the eardrum. The continually changing force imparted by the wave over the area of the eardrum, hence continually changing pressure, is interpreted by the brain to produce the sensation that we call sound. This process can be seen in figure 1-1, which outlines the basics of the human auditory system. It should be noted that in this figure the arrow directing the flow of information to the brain omits the action of additional parts of the system, such as the malleus, incus, stapes, oval window, and the cochlea [18].

It therefore follows from the above, that the perceived properties of a given sound are completely defined by the change of air pressure over time, experienced at the eardrum of an observer. If we consider that a particular sound has a finite duration of d seconds, say, we may represent the sound as a graph of pressure against time, over the interval $[0, d]$. Figure 1-2 shows a graph of pressure in Pascals[5] against time in seconds, over the previously stated interval. This graph, as such, completely describes

[5]One Pascal (Pa) equals a pressure corresponding to the force of 1 Newton over 1 m^2. The human auditory system allows for the perception of tones with sound pressures in the range of 0.00002 Pa to 20 Pa. The highest pressure in this range is otherwise known as the threshold of pain.

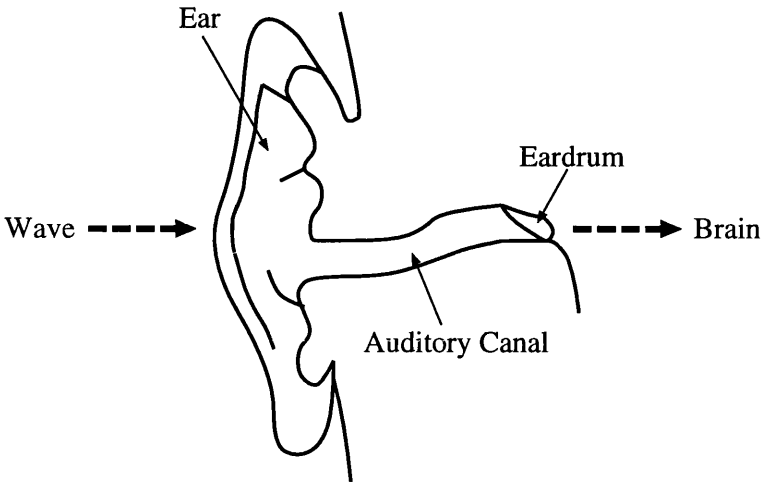

Figure 1-1: *An Outline of the Human Auditory System*

the sound used to generate it over this period. It can be seen from this graph that the pressure, or sound pressure as we shall call it, varies over time within the range of 1 Pascal and -1 Pascal. A negative sound pressure corresponds to a force imparted on the eardrum in a direction away from it, i.e. along the auditory canal towards the ear. This is caused by the air molecules moving in this direction at this point, resulting in a sucking action.

A sound pressure against time graph, as in figure 1-2, can be seen as representing a function of time, yielding a sound pressure value as output. Hence in our example we have a function of the form $f : [0, d] \longrightarrow \Re$, where $\Re$ represents the real numbers from which we take our sound pressure values. This type of function we call, not surprisingly, a pressure function, and thus when we talk of synthesizing a sound, we mean the construction of a representation of its pressure function.

1.2.2 The Digital Representation of Sound

As we have already seen, sound exists as the continual fluctuation of pressure at a given point in a given medium. We have therefore shown that any sound can be represented as a graph of sound pressure against time, and thus as a pressure function. These representations, like the sounds they represent, are continuous. Digital computers, on the other hand, operate on non-continuous or discrete data. Hence, if a computer is to be used to manipulate, synthesize, and output sounds we must be able to convert from the continuous representation of the pressure function into a discrete representation that can be used by the computer. In addition, for the purposes of output, we must be able to convert this discrete representation back into a continuous one, to drive a

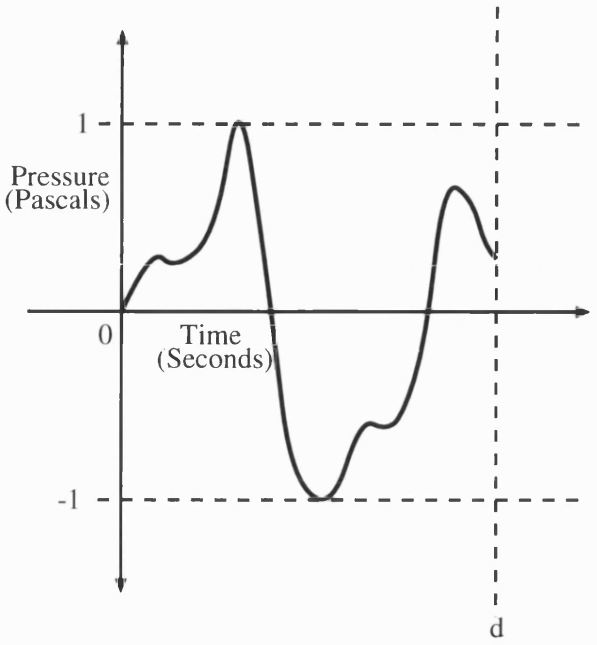

Figure 1-2: *An Example of the Pressure against Time Graph of a Sound*

loudspeaker for example.

The work of Max Matthews at Bell Laboratories in 1958 succeeded in solving these two important problems in digital sound representation. The two processes produced have become known as *Analogue to Digital Conversion* **(ADC)** and *Digital to Analogue Conversion* **(DAC)**. We will, in the following text, examine briefly each of these processes in turn.

Analogue to Digital Conversion

Let us first assume that we have an analogue, i.e. continuous, signal which describes a pressure function over an interval of time $[0, d]$. Analogue to digital conversion seeks to represent this function discretely by producing a sequence of values, or samples, generated by evaluating the pressure function at fixed intervals in time. If we wish to produce a sequence of n samples then we must sample at intervals of $d/(n-1)$, from 0 seconds up to d seconds. We are therefore taking samples at a rate of $(n-1)/d$ per second. This value of samples per second is called the *sampling rate*, which is measured in Hertz (Hz)[6]. If we have a pressure function of the form $f : [0, d] \longrightarrow \Re$, and wish to take n samples, hence at a sampling rate of $S = (n-1)/d$, then each sample s_i is given by the equation $s_i = f(i/S), \forall i = 0 \ldots n-1$.

[6]Hertz are a measure of cycles per second. One Hertz, written 1 Hz, indicates one cycle per second.

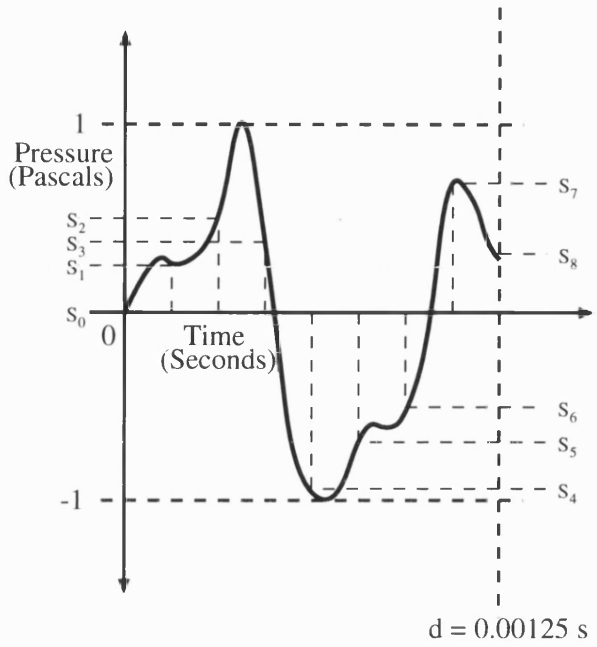

Figure 1-3: *Analogue to Digital Conversion (ADC)*

Figure 1-3 shows a pressure function sampled nine times in its duration of $d = 0.00125$ seconds, and hence at a sampling rate of $8/0.00125 = 6400$ Hz, or 6.4 kHz. The values of the samples are therefore given by $s_0 = f(0/6400) = 0$, $s_1 = f(1/6400) = 0.25$, etc. Table 1-4 shows the values for all the samples generated. These pressure values will more commonly be converted into integers of a fixed storage length, say 16 bits, before being passed from the analogue to digital converter to the computer. Hence via the process of **ADC**, the digital computer is able to gain a set of discrete data, representing the sound, this being the sampling rate, coupled with the in order sequence of converted samples produced.

Digital to Analogue Conversion

Digital to analogue conversion seeks to convert the digital representation of a sound as seen above, i.e. the sampling rate and the sequence of samples, into its analogue equivalent. This is achieved by using the sequence of samples as the skeleton of the sound and smoothly interpolating between each of them, as is shown in figure 1-5, where each value such as S_1, in the figure, is a sample. This results in a continuous signal which can be converted to current to drive a loudspeaker.

Sample	*Time t, milliseconds*	$f(t)$, *Pascals*
s_0	0.00000	0.00000
s_1	0.15625	0.25000
s_2	0.31250	0.50000
s_3	0.46875	0.40000
s_4	0.62500	-0.95000
s_5	0.78125	-0.65000
s_6	0.93750	-0.50000
s_7	1.09375	0.65000
s_8	1.25000	0.30000

Figure 1-4: *Table of Samples Generated via ADC*

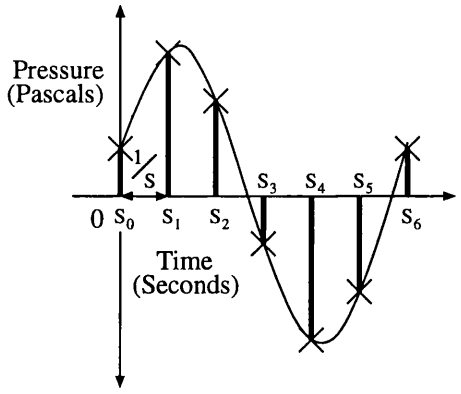

Figure 1-5: *Digital to Analogue Conversion (DAC)*

1.2.3 The Fundamental Problem of Computer Sound Synthesis

In his 1963 article in *Science* entitled "The Digital Computer as a Musical Instrument", Max Matthews pointed out that "Sound from Numbers" was a completely general way of synthesizing sound due to the boundedness of the bandwidth and dynamic range of human hearing[7] [21]. In 1969 Matthews went on to outline the main problems of sound synthesis is his book "The Technology of Computer Music". In it he wrote

> The two fundamental problems in sound synthesis are (1) the vast amount of data needed to specify a pressure function - hence the need for a very fast program - and (2) the need for a simple, powerful language in which to describe a complex sequence of sounds.

The first of these problems has to some extent been solved by the increase in the speed of hardware. At least it has been solved in the sense that the problem in the performance of a particular synthesis technique has tended to be solved by the advance of technology[8], leading to the development of more powerful computer music workstations [51].

The second of these problems has yet to be solved and, it has been claimed, can never be. This is, ironically, due to what would seem to be the main asset of the digital representation of sound, namely its complete generality. Strictly speaking, however, it is not the generality itself that is the problem, rather the large quantities of data required to achieve it. It is unlikely that anyone would wish to type in manually something of the order of 192,000 bytes of data for just one second of sound[9]. Therefore waves must be "synthesized" using recordings, functions, or physical models etc. This is to decrease the amount of data needed to specify a pressure function. The problem with this, however, is that it must inevitably lead to a loss of generality. It is the goal of all research into computer sound synthesis, with the inclusion of the work of the author, to provide a solution to this problem. Past and present proposed solutions will be examined in the following chapter, and from the described failings a superior solution will be given.

[7] Provided that sampling occurs at a rate above the Nyquist frequency, to avoid aliasing in the frequency domain [36]

[8] As an example one might consider the increase in performance of the Csound program in recent years, such that it is now capable of doing sound synthesis in real-time.

[9] This is in the case of stereo sound at a sampling rate of 48 kHz, and with sixteen bit samples. A sampling rate of 44.1 kHz is the standard for compact disc (CD) storage, which is considered adequate for natural sounds.

Chapter 2

Sound Synthesis Systems and Techniques

2.1 Sound Synthesis Techniques

2.1.1 Introduction

The need to reduce the number of parameters required to specify a pressure function has lead to the splitting of the problem of synthesizing sounds into two distinct areas. The first problem is to find a method by which we might specify a time dependent pressure function, the qualities of which are determined by a manageable number of parameters. The second problem is to provide a means of specifying how these parameters, supplied to the function, change over time. In more traditional terms the problems outlined above are, as we shall see more clearly further in the text, those of specifying an orchestra of instruments, and a score for the instruments to play. Therefore, for convenience, we will call the two problems of specification: *instrument specification* and *score specification* respectively[1].

2.1.2 Instrument Specification

When specifying an instrument, we are defining, as we have said above, a function which takes as arguments, time and a fixed number of additional parameters, and will produce a sound pressure as output. Let us suppose that our function, in addition to time, takes a further n parameters which we call p_i, $\forall i = 1 \ldots n$, with each $p_i \in P_i$ (the set of possible parameter values). Then our function f will take the form f :

[1]This split is not strictly necessary to produce large scale compositions. Indeed it has been shown that is possible to create pressure functions directly to produce a composition, by the use of recurrent iterated function systems [15] for example.

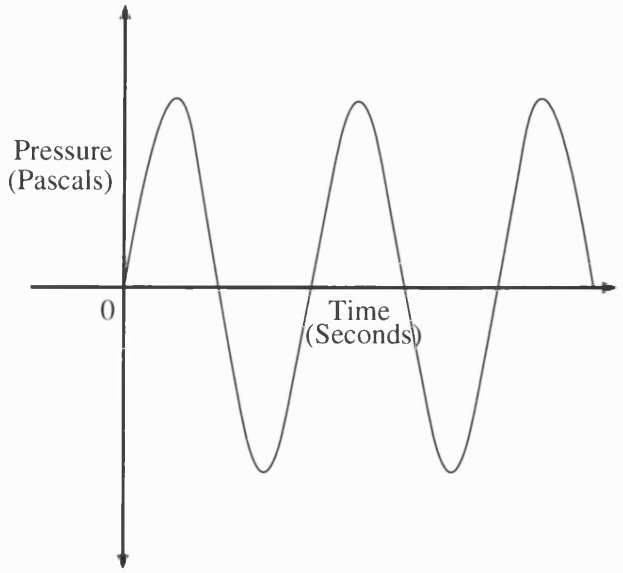

Figure 2-1: *The Output of a Simple Sine Wave Instrument*

$\Re \times P_1 \times \ldots \times P_n \longrightarrow \Re$, if we take our values of time and sound pressure to be real numbers.

As an example let us suppose we wish to define a function which produces a simple sine wave, as in figure 2-1, as output. We will allow our example sine wave instrument to depend upon two parameters. These will simply be those of frequency and amplitude, which we will define as real numbers. Our instrument is therefore defined by a function of the form $f : \Re \times \Re \times \Re \longrightarrow \Re$, with $f(t, f, a) = asin(2\pi ft)$, where t is time in seconds, f is frequency in Hz, and a is amplitude in Pa.

The above example is rather simple and as result the output of our instrument, a pure sine wave, is not of much musical interest. Research carried out in this field has, therefore, centred on the search for methods of synthesis which produce timbres that are complex enough to be musically interesting, but are specified by a set of parameters that is sufficiently small, and meaningful, to allow canonical control of the output. This has been achieved in many ways ranging from simple frequency modulation (FM) [17], spectral analysis [11], physical modelling [37, 46], neural networks [38, 57], and chaotic oscillators [9], to name but a few.

Unit Generators

After outlining the two major problems in digital sound synthesis Max Matthews went on to state the following regarding the problem of pressure wave specification, and hence that of instrument specification.

> The composer would like to have a very powerful and flexible language in which to specify any sequence of sounds. At the same time, he would like a very simple language in which much can be said in a few words, that is, one in which much sound can be described with little work. The most powerful and universal possibility would be to write each of the millions of samples of the pressure wave directly. This is unthinkable. At the other extreme, the computer could operate like a piano, producing one and only one sound each time one of the 88 numbers was inserted. This would be an expensive way to build a piano.

It was this problem that Matthews intended to solve with the concept of unit generators, which he first introduced with the Music V program [50]. The input to a unit generator takes the form of numeric parameters and control, or audio, signals. Each type of unit generator provides a particular functionality which determines how the inputs are dealt with to produce an output. For example, an "output" unit generator takes a single signal as input, which it writes to a sound file. Unit generators can be viewed, therefore, as building blocks from which many synthesis techniques, of differing complexity, can be constructed. Typical unit generators found in the Music V program were an oscillator, filter, adder, output, and so forth. Figure 2-2 demonstrates how FM synthesis may be constructed from unit generators, namely two oscillators and one adder. In the diagram the value F_1 is the carrier frequency, A_1 the output amplitude, F_2 the modulator frequency, and A_2 the modulator amplitude which is redundant, and always equal to one[2].

A cited benefit of unit generators is that it provides a mechanism by which the composer can choose to take any position between the two extremes stated in the above problem, i.e. between manual sample writing, and piano emulation. Matthews himself has said that

> In a given instrument, the composer can connect as many or as few unit generators together as he desires. Thus he can literally take any position he chooses between the impossible freedom of writing individual pressure-function samples and the straightjacket of the computer piano,

While it is certain that the composer can, with unit generators, choose the level of complexity of his/her instrument definitions, it may be that in many cases the model

[2]It should be noted that the resulting function for the output of this diagram does not match the original functional definition of FM synthesis. Both are, however, commonly accepted implementations of FM. The differences between these, and other implementations are discussed in Frode Holm's paper "Understanding FM Implementations : A Call for Common Standards" [17].

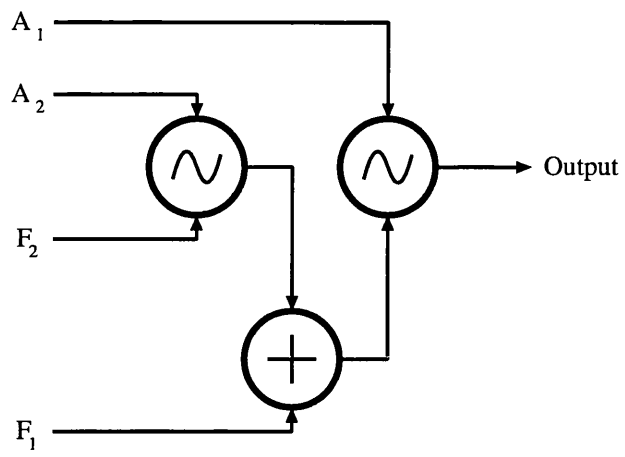

Figure 2-2: *FM Synthesis Constructed from Unit Generators*

is too low-levelled to allow the easy definition of instruments of significant complexity. The likelihood of this occurring is dependent upon the size and scope of the set of unit generators from which one is to build an instrument. Unit generators form the building blocks for instrument definitions in the Csound [64] language, the popularity of which is evidence of the strength of the unit generator model. In Csound, however, the set of unit generators is extensible so that generators with more complex functionality, or perhaps the functionality of many generators combined, can be added to the system. Given these new unit generators the problem of specifying a complex instrument can be significantly lessened. In such circumstances, therefore, the unit generator model provides an excellent tool for instrument specification.

Instruments specified with unit generators will obviously have a set of inputs which completely control the output that is to be produced. The unit generator model, however, says nothing about how the values that these inputs take, during the composition, are to be specified. That is to say that the model does nothing to solve the problem of score specification, which we will examine in the following section.

2.1.3 Score Specification

In order to consider the problems associated with the specification of scores, it is firstly important to establish what it is we mean, in the context of this thesis, by the term *score*. We have already seen above that we specify an instrument as a function of a fixed number of parameters, and time, which evaluates to produce a value of sound pressure. The resulting sound generated by this instrument is thus defined by the values of these parameters at each moment of time in the composition.

As an example let us consider a very simple instrument defined by the function

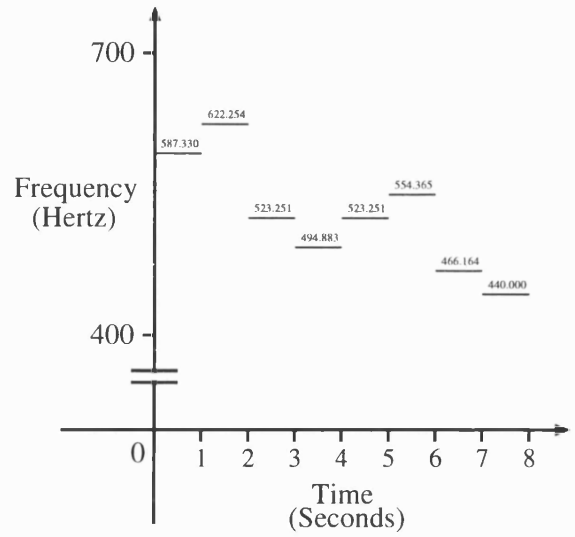

Figure 2-3: *Frequency against Time Graph*

$g : \Re \times \Re \longrightarrow \Re$, evaluated as $g(t, f)$, where f is frequency in Hertz, and t is time in seconds[3]. We will suppose that the time for the composition, in which our simple instrument takes part, runs uniformly from 0 seconds, to d seconds. Hence we may completely specify the behaviour of our simple instrument by setting $f = s_f(t)$, where $s_f : [0, d] \longrightarrow \Re$, and thus the value of our instrument at time t is given by $g(t, s_f(t))$.

In general, however, a particular instrument is specified by a function of the form $g : T \times P_1 \times \ldots \times P_n \longrightarrow \Re$, where $P_1 \ldots P_n$ are sets from which each of the instruments parameter values is taken[4], and T is the set of time values. Therefore, to govern the parameters of the instrument we have functions $s_i : T \longrightarrow P_i, \forall i = 1 \ldots n$, such that at time t, the value of the instrument is given by $g(t, s_1(t), \ldots, s_n(t))$. And so the n-tuple of functions $(s_1, \ldots, s_n)$ can be said to describe the behaviour of the instrument for the duration of the composition in question. A *score* for this instrument is, therefore, a representation of this n-tuple[5].

As an example let us suppose that our simple instrument, with its single parameter of frequency, is determined by the frequency function $g : [0, 8] \longrightarrow \Re$, as shown in figure 2-3. Clearly figure 2-3 is a score for our instrument as it is a representation of a function which determines the value of the instruments frequency parameter over time.

[3] We might consider this instrument to be a simple sine wave instrument with fixed amplitude.

[4] A particular parameter value may in fact be a set of many values. An instrument such as a piano may take as a parameter of frequency a set of notes which are to be played simultaneously.

[5] It should be noted that a score for a combination of instruments is constructed from the individual scores for each instrument. A string quartet for instance can be viewed as a single instrument which takes as parameters all of the parameters of the members of the quartet. The individual scores in summation form a score for the composite instrument, in this case the quartet.

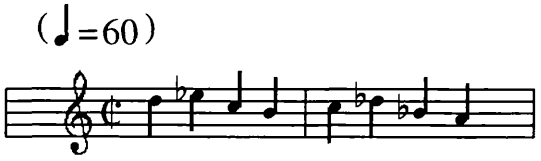

Figure 2-4: *A Frequency Function in Common Western Notation.*

Figure 2-4 shows a representation of the function in common western score notation.

On examining these two representations of a single musical idea, it is clearly the case that the graph representation is the more general. In the graph representation we can describe any frequency in Hertz given by a positive real number. The traditional score however can only deal with frequency changes in steps of semitones[6]. However, for this particular musical idea the traditional notation is to be preferred since the representation is easier to specify. The frequency time graph requires notes, and beats to be translated into frequencies in Hertz and time in seconds respectively, a process which is tedious manually. In a sense, therefore, the graph representation is too general for this particular musical idea. If, on the other hand, we had wished to notate an idea which consisted of microtonal changes in frequency the traditional score would be too restrictive; indeed the specification would be simply impossible using traditional notation. For this particular idea, therefore, the traditional notation system is not general enough, whilst the graph representation is perfectly usable, though not necessarily ideal.

With the above examples we are uncovering a problem in score specification analogous to the problem stated by Max Matthews with respect to the specification of instruments. The inherent problem in score specification is that all score representation systems are either too general, or not general enough [4]. More accurately, for any particular score representation system there will exist musical ideas for which its use is impossible, i.e. for which it is not general enough, or there will exist musical ideas for which it is too general, in the sense we have given above. These two properties are not mutually exclusive since it is possible, in fact quite likely, that for a particular score writing method there will exist musical ideas for which it is too general, and musical ideas for which it is not general enough.

Although this problem is akin to the problem of instrument specification in many respects, it differs in the fact that, for the most part, researchers have largely neglected it in favour of concentrating on how to define instrumental timbres. This is rather odd, since the success of a sound synthesis program must depend upon how well it solves

[6] Assuming that the instrument used to play the score is tuned in the traditional manner, for example a violin with its strings tuned to G, D, A and E in the usual octaves.

the fundamental problem of sound synthesis, as stated previously. This, in turn, is dependent upon the adequacy of its solution to both the component problems of the fundamental problem. Hence, in order to measure the success of a particular sound synthesis programs, we must examine the adequacy of its solution to the problem of instrument specification, and the problem of score specification. This we will do for a number of sound synthesis systems, past and present, in the following section.

2.2 Sound Synthesis Systems

Computer-based sound synthesis systems can be classified into three separate types. These types we will call language-driven systems, batch (off-line) systems, and experimental (on-line) systems. Language-driven systems typically take the form a tool-kit, or library, of executable programs, and computer code, dedicated to sound synthesis. As such they are different to both batch and experimental systems since these two approaches are essentially data driven[7]. Batch, or off-line systems, are characterized by the process of specifying a composition in some language, running the program to "compile" this specification into a sound file, listening to this output, and repeating the process if necessary. Experimental systems allow users to compose via experimentation with sound, and unifying the results into a composition. As such a composition will evolve through this experimentation, rather than having been largely precomposed, i.e. before using the computer, as is often the case with batch systems.

2.2.1 Language Driven Systems

Language-driven systems supply tool-kits, and libraries of functions, as add-ons to an already existing language implemented in a compiler or interpreter. The composer is intended, therefore, to write his/her own programs with the aid of these add-ons that when executed will produce the output, usually a sound file, that is required. Figure 2-5 shows a portion of C code which outputs a tone produced via FM synthesis[8]. The arguments to this function are therefore values representing the amplitude, carrier frequency, modulator frequency, and duration of the resulting wave. This functionality is made possible by the inclusion of the functions *open_sound*, *write_sample*, and *close_sound* which are taken from a C audio library written by the author of this thesis. As such this section of program is typical of code which makes use of a language driven package.

[7] It is possible, however, for a data driven system to implement a language as part of its data input.

[8] In this case the FM synthesis implementation accords with its original functional definition, which was not the case with the implementation, seen earlier in text, using unit generators

```
int play_fm(double amplitude,double carrier,double modulator,double duration)
{
  SOUND *output;
  SAMPLE out;

  double t,step;

  step = (double)1 / 48000; /* Time separation between samples */

  open_sound(&output,PLAY_ONE_AT_48000,""); /* Open mono audio output */
                                            /* at 48 kHz */
  for (t=0.0;t<duration;t+=step)
    {
      out = (SAMPLE)(amplitude * sin(2 * PI * carrier * t
     + sin(2 * PI * modulator * t)));

      write_sample(output,out); /* Calculate and output FM sample */
    }

  close_sound(&output); /* Close audio output */

  return 1;
}
```

Figure 2-5: *A Language Driven Approach to FM Synthesis*

Cmix [26, 50], written by Paul Lansky at Princeton University, consists of a library of C functions and sound manipulation programs for the purpose of sound synthesis. It is a more developed version of an earlier program called MIX, also written by Lansky. The MIX program, which was written in FORTRAN, was essentially a program for mixing several independently specified voices into a unified composition. Cmix evolved as the MIX program was rewritten in C, hence Cmix, and ported to more powerful UNIX workstations.

Instrument definitions in Cmix are simply C functions, which when called with their arguments supplied, write[9] an output to a particular sound file specified in the score. To produce a Cmix program the user must compile their instrument definitions with a C compiler, and link in the Cmix library. Cmix programs are, therefore, executed at the command line with a score file being communicated to the program via the standard input stream i.e. for a Cmix program called *fm_output*, with a score file called *fm_score*, the UNIX command would be `% fm_output < fm_score`.

Cmix score files are specified in a C-like language called MINC[10], which is part of the Cmix system. A MINC score will usually begin by specifying the names of the sound files to be used as output[11], and the assignment of generators for the function tables. There then follows any definitions of global variables that are to be used in the score. Finally the sound generating instrument calls are specified. The link between the note commands in the score, and the instrument functions defined in the instrument program, is specified in the instrument definition code via the specification of the *profile* function, in which the user must execute functions which establish the links.

Both the instrument specification system, and score specification system in Cmix are very powerful. The score writing mechanism, being essentially the C language coupled with the Cmix library, is highly general and therefore any synthesis technique can be implemented provided that the user can program it. The C language is fairly complicated, however, in comparison with the instrument specification languages of other systems, which we will see later. There is, therefore, a much slower learning curve, and hence a greater length of time required before anything of complexity can be achieved, in the realm of instrument design. In the area of score specification the MINC language is extremely powerful, since it implements many control structures, such as loops, found in normal programming language, but not usually in a score specification language. The fact that MINC is very similar to C allows for the definition of macros,

[9]The operation of the function to write to a sound file can be specified to be more than simple over writing. Cmix also allows for additive writing to sound files, for example.

[10]MINC is a recursive acronym for MINC Is Not C.

[11]It is normally necessary to create these files before they are added to, in order that the correct header information is present.

and global variables, which gives the user some control over the level of generality at which to work. For example the user could define macros which produce chords, or other more complicated structures, and concern himself with the control of these, rather than simply specifying a sequence of notes. The similarity to C, however, again implies a reasonable quantity of learning time, which must be undertaken as before.

These comments regarding Cmix, lead to the conclusion that it, like similar language driven systems such as Common Lisp Music [61], is not really a stand-alone sound synthesis program, but is rather a tool-kit for the construction of sound synthesis packages. Its powerful library of functions, and tools, allows for a person of skill and expertise in programming, and sound synthesis, to construct a sound synthesis system that is usable by people with less ability, and who do not wish to spend a lot of time acquiring it, in order to achieve the sounds they desire. Such systems may be very similar to those we will discuss in the following section.

2.2.2 Batch Systems

Batch systems are the kind most traditionally associated with software sound synthesis. By use of the term batch, we mean to consider systems in which one proceeds by writing the specification of a composition, in some language, and then running a program on this specification to produce the corresponding sound file. This sound file can then be listened to by running a "play" program on it. The specification may by modified if it is not valid, or the output is not as desired, and the process is repeated. We are, therefore, in the realm of what we might call sound, or audio, compilers.

The Music V program [50], developed by Max Matthews in the late 1960's, was the first of such systems to be widely used[12]. Due to its popularity, many of the aspects of the Music V program are to be found in the software sound synthesis programs of today. Most notably the input to the Music V program is in two parts; firstly a set of instrument definitions specifying an orchestra, and secondly a set of note lists for each instrument in the orchestra. Instruments are defined by the combination of unit generators, such as oscillators, linked to an output generator, to produce an instrument's sound. Scores, in Music V, are specified in terms of note lists for each instrument in the orchestra. Each note in the list consists of an in order sequence of parameter values, such as note start time, duration, frequency, amplitude, etc for the instrument given by the instrument number parameter. The Music V style of sound synthesis software has become very popular resulting in the production of systems

[12]This is most likely due to the fact that it was the first to be written almost entirely in a high-level language, namely FORTRAN, as opposed to assembly language. This made it easily portable to other platforms.

```
;       Orchestra Specification

        sr = 48000                  ; Set sampling rate to 48 kHz.
        kr = 4800                   ; Set control rate to 4.8 kHz.
        ksmps = 10                  ; Number of samples per control
        nchnls = 1                  ; cycle is ten, and number of
                                    ; output channels is one.

        instr   1                   ; Set amplitude to increase from
kamp    line    0,p3,10000          ; 0 to 10000 over period p3.
asig    oscil   kamp,cpspch(p5),1   ; Get oscillator value, and output.
        out     asig
        endin
```

Figure 2-6: *An Example Csound Orchestra File*

similar to Music V in more recent years. The most notable of these is the Csound program. Due to the close relationship between these two programs, the comments that we make concerning Csound, in the following text, will also apply, to a large extent, to the Music V program and to the similar Cmusic [50] program.

Csound [64], developed by Barry Vercoe at the MIT media lab, can be said, with some confidence, to be the most popular of sound synthesis programs in use at the present. This is largely due to its portability, as it is written in the language C, its speed, and its upward compatibility with the earlier synthesis program Music-11, also developed by Vercoe at MIT. Like the Music V program, Csound generates audio output from an instrument specification file, called the orchestra file, and a score specification file, called the score file. Again, as with Music V, instruments are specified using unit generators. Mathematical functions and arithmetic operators are supplied, and for additional control, *if*, and *goto*, statements may be used to give extra complexity to the instruments defined.

An example of an orchestra file can be found in figure 2-6, in which a single instrument is defined which produces a sinusoidal output at a given frequency, and a uniformly increasing amplitude. From this example we see that the Csound score language is similar in some respects to an assembly language for writing machine code. This correspondence is further revealed by the fact that like assembly language, the score language is easy to learn, as there are very few commands, but it is also difficult to construct anything of considerable complexity, and equally difficult to debug.

An example of a Csound score file can be seen in figure 2-7, which is one possible score for the orchestra file, consisting of a single instrument, shown in figure 2-6. The

```
;       Score Specification

f1 0 256 10 1                          ; Use Gen10 to fill function table 1

i1 0   0.5 0   9.02                    ; D (8-ve above middle C) for 0.5 seconds
i1 0.5 .   .   9.04                    ; E
i1 1   .   .   9.0                     ; C
i1 1.5 .   .   8.0                     ; Middle C
i1 2   .   .   8.07                    ; G

e                                      ; End score file
```

Figure 2-7: *An Example Csound Score File*

most important part of a Csound score is formed by lines which begin with an instrument number, for example i1, as seen in the sample score. This indicates that the note described in the line is to be played by the instrument defined in the orchestra file as instrument number one. The values in such a line are called a *p-fields*[13], which are identified by the terms p1, p2, p3, etc along the line, beginning with the instrument number. For example in the first note defining line in figure 2-7, p1 = i1, p2 = 0, p3 = 0.5, p4 = 0, and p5 = 9.02. These terms correspond to those used in the definition of the instrument. Hence in the score we can see that p2 is the time at which the note begins, p3 is its duration, and p5 specifies its pitch, which in this case is given in terms of octave number, given as 9 or an octave above middle C, and a semitone value 02 or D. This value of pitch is converted into cycles per second i.e. Hertz, as required by the oscillator unit generator, via the use of the function *cpspch*, as seen in the instrument definition.

The Csound program is executed at the command line with the orchestra file name, and score file name, as arguments. Thus, if we have the orchestra file *demo.orc*, and score file *demo.sco* we might compile them to produce a sound file using Csound with the Unix command `% csound -A demo.orc demo.sco -o demo.aiff`. Upon execution Csound sorts the note events in the score into chronological order. The sorted score is then used to drive the orchestra described in the orchestra file. The result of the orchestra's performance of the score is a sound file called *demo.aiff*, in the AIFF [63] file format. The sound file produced, assuming no errors occurred in compilation, can then be played by executing a "play" program, again from the command line. The user

[13]The term stems from the fact that each value is a parameter for a particular instrument. Therefore each term is a parameter field, and thus p-fields.

must then, if necessary, make adjustments to the orchestra and score files and repeat this process until the desired sound file is produced.

The Csound orchestra specification language, whilst being rather low-level, provides a powerful tool for instrument definition, particularly since the set of unit generators available can be extended, though this does require a knowledge of C, to provide greater complexity. The real problem with Csound is its score file format. The note list format, where each note is specified by a sequence of p-fields, is extremely general. Indeed it is so general that the specification of many rather trivial, and commonplace musical ideas, such as a simple piano piece, which is represented more easily in common western notation, requires a great deal of tedious effort. The score system is then rather too general for many types of musical ideas. However the power of the instrument description language, the speed of current implementations of the program, and the generality of the score language, makes the Csound language ideal as a kind of virtual machine architecture that higher level sound synthesis programs can compile down to, and then produce a sound file using the Csound program.

An example of the approach mentioned above is the use of MIDI to generate Csound via the MIDI2Csound program, and in general the use of C programs to generate Csound output from higher level descriptions. One such program is called *scot*, which is part of the Csound distribution. The scot program allows the specification of scores in the scot score description language. This language is for the specification of scores in a similar fashion to common notation, and as such provides mechanisms for specifying key signatures, transpositions, and so forth. It is, however, a text based language which is far from easy to learn, or to use. Ideally one would wish to specify a traditional score using some graphical editor, which could translate the graphical score into the scot language, for use with the scot program, and finally Csound. The length of this chain of compilation down to the Csound representation is evidence of the low level nature of the Csound program, and thus its inadequacy as a sound synthesis system where its input files are specified manually. Like machine and assembly code, it is best seen not as a language in which to write, but as a target code for the compilation of higher level representations.

2.2.3 Experimental Systems

In the early days of sound synthesis the speed of the available hardware was not sufficient to produce the digital representation of a pressure function, of reasonable complexity, in a short period of time. That is, a period of time short enough to enable the user of a sound synthesis package to proceed in an experimental fashion. By this we mean the composition of a musical work via experimentation with sonic media made

available through the system. Instead of this, the lack of performance, meant that computer based compositions were largely precomposed, with the computer merely being used to realize them, in much the same way as a conventional instrument is used in the performance of a score.

Experimental computer music (ECM), taking advantage of the increased speed of modern hardware, seeks to allow the user to compose music via the process of experimenting with sound available through the computer, manipulating, editing, and unifying the results into a composition [40].

The UPIC system [53, 31] is an experimental computer music system which provides fast audio output from scores specified by drawing on a graphics pad. In this manner all, aspects of a particular composition from the low-level waveforms specifying a timbre, to the specification of parameter values, such as frequency, are specified as a number of superimposed graphs of the property in question against time. Figure 2-8 shows part of an UPIC score. This figure shows what is called, in the system terminology, a *page*, specifying frequency against time, where the frequency axis runs vertically, and the time axis horizontally. The calibration of each axis can be determined by the user either manually, or from a set of prestored tables, such that the frequency axis may run in steps of semitones for example, from pixel to pixel up the page. Portions of a page may be enlarged so that the work can be more detailed in places, whilst the pages themselves can be overlapped arbitrarily in time to form a composition.

The UPIC system is essentially, therefore, a painting package which provides a mapping from its images into sound[14]. This similarity is not wholly surprising when one considers that a prime motivation behind its design was pedagogy. This is clear from the words of the system's initiator Iannis Xenakis who has stated [30]

> I want to have a tool for myself and other people that will be general enough to be used in pedagogy, so as to bridge the gap that exists between "normal" people and contemporary music developments. If anybody is able to use such a machine, it will heighten the awareness of the average person who will then be involved in composition also. This makes a much more homogeneous environment for music.

The quest for pedagogy has, however, lead to the adoption of a score representation that is general enough for such use, but is rather too general to represent easily many types of musical idea. Even with the ability to calibrate the axes of the graph representation, it is still a two dimensional graph and is hence only capable of specifying two

[14] In many ways this is similar to the process of audification, or sonification. Systems which implement this process provide a mapping from data in n dimensions onto a sonic representation [25, 24], by mappings tuples onto pitch, amplitude, timbre, and other musical properties.

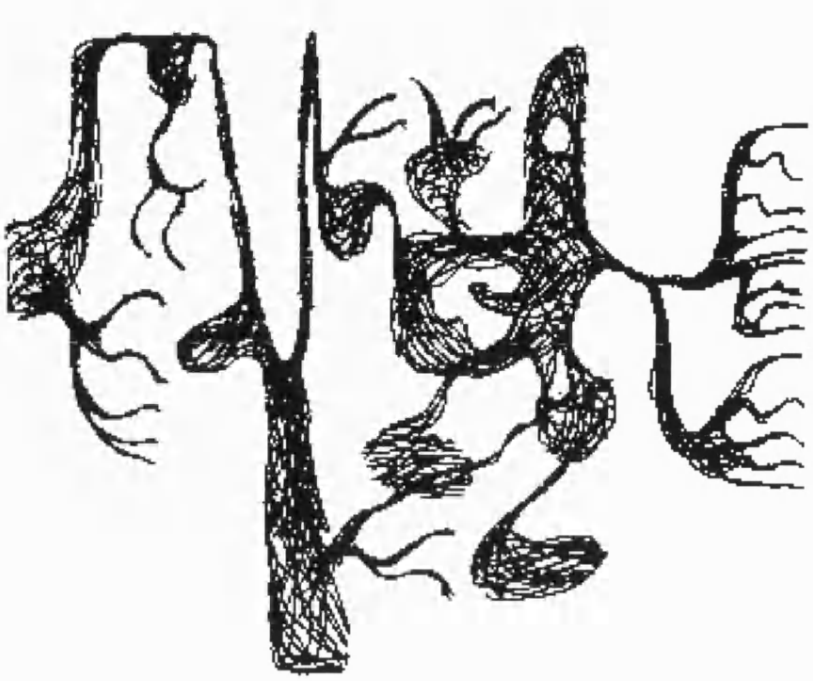

Figure 2-8: *A Page of an UPIC Score Showing a Plot of Frequency against Time*

classes of parameters at once. In many cases, as in common notation, we may wish to specify pitch, amplitude, ornamentation, all at once against time. The UPIC system is therefore inadequate as a system for experimental computer music, for although it supplies a short turn-around time from the specification of a score to the production of its audio realization, the specification method is far too general for many types of musical ideas.

The DMIX system [42], written by Daniel V. Oppenheim at the Center for Computer Research in Music and Acoustics (CCRMA), is a computer-based environment for experimental computer music and music performance. According to Oppenheim his

> main motivation was to design an easy-to-use and yet flexible and general environment that has a uniform user-interface, is easily extensible, and is independent of any synthesis hardware or host computer.

DMIX itself consists of graphic and text based tools, along with tools for real time editing of musical ideas (i.e. while they are being played), performance tools (for the use in composition and performance), general tools such as functions, and a music representation which is directly changeable by the user. The use of multiple views, or

representations, allows for the specification of scores, which at the lowest level are lists of events which trigger the action of objects in DMIX, can be achieved, to an extent, using an appropriate representation. The representations available are, however, either text or graph based which are not necessarily ideal.

The most powerful aspect of the DMIX system is the paradigm of *slappability*, which is used for applying tools, and representations to each other, and themselves. For example a representation of a section of a score may be *slapped*, via grabbing and dropping, onto the algorithm description tool, called QUILL, resulting in the score being represented in algorithmic form, which can then be modified.

The main problem with DMIX appears to be concerned with its method of constructing more complex pieces from the initial results of experimentation. This is achieved via the mixing in time of the elements generated using the tools available. The changing of other aspects of these elements, such as pitch, amplitude envelope, are achieved by the application of what are called *modifiers*, which modify, i.e. change, these elements in the desired fashion. This is a problem since during experimentation we may come to a point at which our results are not as desired, and hence wish to go back to a previous state, from which we believe we made a mistake. As the elements we have used have been modified in some way, we must either undo all these modifications, or recompose the elements we began with, which hinders the process of experimental composition.

The DMIX system gives the impression of being a very powerful and general sequencer, in the style of a MIDI sequencer, with the difference being that it deals with more general types of events, rather than simply MIDI control. Indeed one of the main objects in the system is called an *echo* which acts in a similar fashion to a MAX *patch*. This similarity is, perhaps, not surprising since a goal of the DMIX project was to unify the compositional, and performance aspects of computer music. The addition of SHADOW [39], and LeNNY [41], both tools for adding performance nuances to scores, would seem to indicate a tendency of the system towards composition via the addition of parts generated by ever more complex, and detailed, performances. Whilst performance is important for the experimentation with ideas, and final production of audio output, attention must be given as to how the results of experimentation can be unified and then experimented with, in a similar fashion, the results of which can easily be rejected, and something else tried, if the result is not a desired. It is in this area that DMIX is not fully adequate.

2.3 Remarks on Computer Sound Synthesis

In this chapter, and that which preceded it, we have shown that the problems relating to computer, or digital, sound synthesis stem directly from the nature of sound, and hence its digital representation. The near impossibility of specifying each of the samples required to represent a pressure function, it was noted, has lead to the splitting of the problem of digital synthesis into two component problems.

The problem of what we called instrument specification was shown to be solved in many different ways, that is with the use of a large array of synthesis techniques. The concept of unit generators was shown to enable a composer to construct a particular synthesis technique using the generators as primitives. In this way a synthesis technique could be produced of the desired complexity, and generality.

The problem of what we called score specification was shown to be founded upon the fact that any particular score writing method will inevitably be either too general, or not general enough, for the representation of particular types of musical ideas.

From our investigations into the problems of computer sound synthesis we uncovered a method by which we might evaluate sound synthesis software. This was shown to be achieved by examining the adequacy of the solutions, that a system provides, to the problems of instrument and score specification. The systems that we analysed in this manner were found to be split into three categories into which they were divided due to their approach to computer sound synthesis. Language-driven systems were shown not to be true sound synthesis systems, but are rather tool-kits for the construction of sound synthesis systems. We will therefore disregard this approach as a viable approach for computer sound synthesis software. Of the remaining two approaches, the paradigm of experimental computer music is to be preferred. This paradigm scores over that of batch systems since an experimental computer music system, equipped with a large array of tools and materials, can be used in the same way as a batch system. Batch systems, on the other hand, cannot provide the functionality of experimental systems in a similar fashion. Therefore the approach of experimental systems is superior since it affords greater flexibility.

The DMIX environment was found to implement many of the properties required of an experimental computer music system. It was shown to provide a large number of input, i.e score description, mechanisms and tools for their manipulation, in particular the powerful paradigm of *slappability*. The production of the source material for further experimentation is, therefore, catered for with considerable success in the DMIX environment. It was found to be less than desirable, however, in terms of the construction, and deconstruction, mechanisms concerned with the manipulation of works that

are the combination of many smaller parts.

The purpose of the first part of this thesis has been to examine the problem of computer sound synthesis and to extract the properties that are required of a computer system for sound and music sythesis. It has identified those properties that as yet have not been supplied by any computer system. The second part of this thesis will describe the theory behind, and implementation of, a system in which these remaining properties have been included, with the intention of showing that a system, which solves all the problems described, is indeed possible.

Part II

A Prototypical System: Theory, Design and Implementation

Chapter 3

Sound Construction Model and Theory

3.1 Introduction

We have now reached the point at which we have shown that the digital synthesis of sound with computers is hindered by a single inherent problem. This is due to the large quantity of numerical data needed to specify a pressure function. As a result the problem of sound synthesis has been in the split into two problems, which we called instrument specification, and score specification. Having made this split it was discovered that each of these component problems suffered from difficulties related to generality.

For the purposes of specifying an instrument, there exist a vast number of synthesis techniques that might be used. However, with the use of unit generators, provided that the set of generators is extensible, we can construct a synthesis technique of the complexity and generality we require. The use of the unit generator model does not, however, mean that research in the field of synthesis techniques is at an end. Unit generators do not provide all synthesis techniques that are possible, but only a way of modelling those synthesis techniques that have so far been discovered. It may also be the case that the use of unit generators to specify the instruments for a composition, say one in which all the instruments are defined using some new and complex synthesis technique, would prove to be more trouble than to simply write a stand alone program implementing this new technique. For situations in which many different synthesis techniques are required, they are a powerful method of specification.

The problem of score specification was shown to be the result of a problem that is inherent in all score types. Namely that a certain type of score may well be perfectly

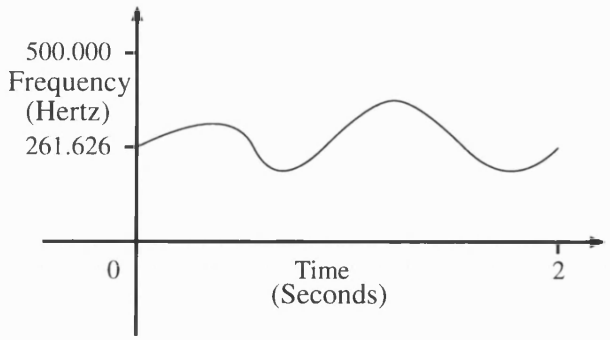

Figure 3-1: *Frequency against Time Graph for the Choral Part*

adequate for a certain type of musical idea, but will inevitably be too general, or not general enough, to represent some other type of musical idea. The solution to this problem appears, at first glance, to be fairly simple. We will, therefore, illustrate this potential solution in the following section.

3.2 A Potential Solution for Score Specification

Let us suppose that we are producing a composition using a computer music system, and that for a section of this work we have a particular idea in mind. Firstly, we wish to have a choral part occupying the lower registers, which rises and falls in pitch, through microtonal intervals. Secondly, above this, in the higher register, we desire a trumpet fanfare, which is thus restricted to a smaller set of notes, i.e. those that are achievable on the modern trumpet. For the first of these parts it appears appropriate to specify frequency with the use of a frequency against time graph, as shown in figure 3-1. This graph representation is, however, too general for the efficient specification of our more traditional trumpet fanfare. For this part it would be more appropriate to use the common western notation, as seen in figure 3-2. This system of notation is, of course, not general enough to specify the microtonal choral part. Hence we have reached a position in which the most appropriate system of score specification for each part is not appropriate for the other. The obvious solution to this problem is to use the most appropriate method for each part, and simply equate the final sound for this section to the sum of the sound for each of the parts. This solution, however, will be shown to be inadequate in the realm of experimental computer music, which we have shown to be the most flexible paradigm for computer sound synthesis.

In order to understand why the above solution is inadequate, we must first consider what it is we are doing when using a computer for the synthesis of sound and music. Let us consider a particular situation in which we might want to use a computer system

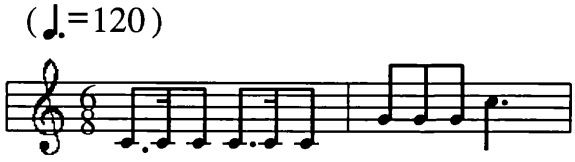

Figure 3-2: *Trumpet Fanfare Part*

for sound synthesis. Suppose that we have before us a score, in common western notation, for a short piano piece that we have written. We now wish to use a computer music package, say Csound, to produce the sound for this composition, using computer generated timbres. At this stage the process of composition is over[1]. The problem that we have here is simply one of setting our composition, in the required format for the system to use. As such it is analogous to an exercise in typesetting. If this situation was mirrored in our two part example composition above, i.e. the scores for both parts are already finalised in the most appropriate system of specification, as shown in the two figures, then we could realise the composition using the suggested solution. That is use an appropriate score writing tool for each part, generate the resulting sound, and the add them to produce the required result.

In an experimental computer music system, however, the situation is quite different. Here we are using the computer system to experiment with ideas, via trial and error, although probably guided by some overall concepts, in order to produce a composition. In such a situation there will exist a number of "primitive" sounds, such as notes, or note lists, and tools for their generation, which may all be considered to be objects in the computer music system. In addition there will be tools for the modification of these primitives, such as score editors and generators, the outputs of which can be applied to the primitives. Finally there will exist techniques for joining the resulting sounds to form larger sonic entities.

As an example of a situation, found in the real world, that is analogous to the above we might cite the use of a child's building kit. As primitives there will be blocks of varying shapes, colour and size. These can be moved about, turned, and fitted to each other to produce larger forms. These larger structures can also be moved, turned, and used as building blocks in similar fashion. Hence with the use of this analogy we see that composition via the paradigm of experimental computer music takes the form of play with the materials that are supplied by the system, and the techniques available to manipulate them. We construct a composition, therefore, via play with available

[1] At least in the realm of note, and hence score, specification, although the choice of timbres, i.e. instrument specification, may not have been decided.

materials and hierarchically assembling the results [22] into larger structures, which may also be the subject of play and experimentation. Thus, in order properly to allow experimentation with sound, it must be possible to manipulate the larger entities that are the consequence of the joining of experimental results, in the same manner as our primitives. In the following text this will be shown to be essential by examining the construction of our example composition under the paradigm of experimental computer music.

Let us now continue with our example composition. We will suppose that we have reached the point at which we have specified the scores for each part of our composition, i.e. the trumpet fanfare, and the choral element, using the most appropriate method of notation. These we now apply to a trumpet, and a choral instrument respectively, to produce entities representing the resulting sound for each score. We now combine these two entities such that the resulting entity represents the combined sound of the two components. Having achieved this, and on listening to the result, we decide that it would be better if the composite, namely the combined sound of the two scores performed by their respective instruments, sounded an octave higher than at present. The question arises as to how we are to modify the composite entity, such that this is achieved. To this problem there appear to be two obvious solutions which are as follows.

1. Change each component score, such that each frequency specified is an octave higher than before, and in general make modifications to the scores to modify the composite.

   There are a number of problems with this solution which render it inadequate. In order to achieve the modification manually a large amount of work may be required from the user. To achieve this automatically the system must understand how to make the appropriate changes in each score writing method, which may produce a large overhead. Further it would make it difficult for users to define their own notation systems, since they must provide the sound synthesis software with methods by which the score may be modified. The main problem, however, is concerned with the deconstruction of a composite sound.

   Let us suppose that in our example the user finally decides after experimenting with the composite that it is not to his/her liking, but still wishes to use the trumpet fanfare. When the composite is reduced to its components the user finds that the trumpet score has been altered, i.e. that it has been raised by an octave in pitch. The score must, therefore, be modified again to get it back to its original state. For this particular example the operations required to restore the score are

not terribly complicated. When the composite has been experimented with, i.e. modified, to a large extend, however, say a crescendo has been applied, part of it has been raised in pitch etc, the component scores will be almost unrecognisable and hence almost impossible to use again. Hence we see that if we were to use this solution the experimentation with composites would quite often result in the components becoming useless, in terms of being used again.

2. Change the instruments such that the pitch at which they play is an octave higher than the frequency parameter supplied to them, and in general modify the instruments in order to modify the composite.

   The problem with this solution is again due to the results of deconstruction. While the scores will remain unaltered the behaviour of the instruments will have changed. In our example both the trumpet and choral instruments will play any note we wish to hear an octave higher than the frequency we specify. This could be dealt with by remembering how these instruments will behave, and hence specify frequencies an octave lower than we want, or change the instruments back to their original behaviour. However, if the composite has undergone a large amount of experimentation, as in the manner described above for example, the instruments will behave in a complex manner which we would be hard pressed to compensate for. It might be possible to reset the instruments such that all parameters as specified remain unaltered, but this assumes that the instrument behaved in this manner to begin with. Even if this is possible it is certainly not desirable, particularly when the composition, and hence the number of instruments and scores, from which it is formed, becomes large.

From the inadequacies of the proposed solutions, described above, we conclude that what is required is a technique for constructing composite sounds, from less complex components, which will allow for the modification of, and hence experimentation with, the composite without affecting the properties of the component sounds from which the composite is built. The problem is, therefore, to find a method for building and manipulating composite sounds such that the above conditions are maintained. A complete, and original, solution to this problem will be given in the following text.

3.3 Representing Composite Sounds

Before we continue further, we will first examine the issue of representing a composite sound using a data structure within a computer. Let us suppose that we have two simple or primitive sounds which we call *Primitive1*, and *Primitive2*. These we

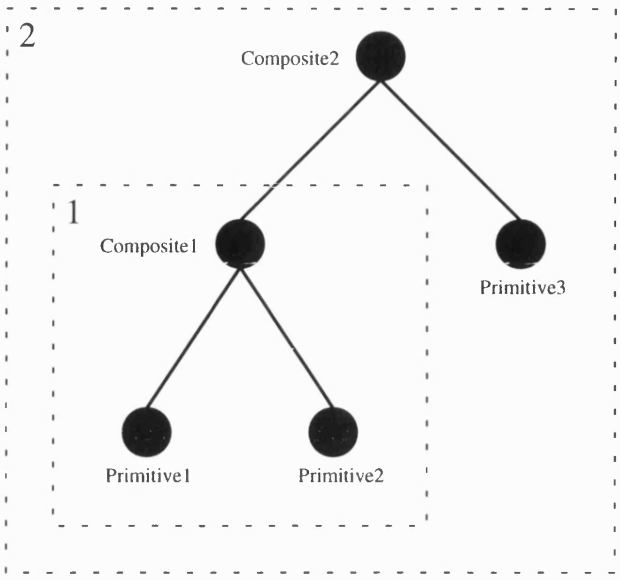

Figure 3-3: *A Composite Sound Represented as a Tree*

combine in some manner to form a composite sound which we call *Composite1*. We then combine this composite sound with a third primitive called *Primitive3*, to form a second composite sound which we call *Composite2*. Each combination of sounds (primitives or composites) forming a composite can be represented by a tree data structure. Figure 3-3 shows the tree representing the sound *Composite2* formed by the above procedure. The subtree, bounded by a box and labelled **1**, represents the composite sound *Composite1*. In general, therefore, a composite sound formed by the combination of a number of other sounds is represented as a tree node, with the sounds from which it is built given as child nodes.

One particular, and notable, example of the use of the tree representation in the construction and manipulation of composite sounds is the TTree [6] of Glendon Diener. This model consists of a binary tree with the horizontal arcs labelled with time delays. The nodes, connected by the arcs, represent musical objects, or *gliphs*, which are activated after the time delays given on the connecting arc. This method forms the basis of Diener's system[2] for notation and composition. In this system the objects at the nodes are notation elements which may be superimposed or distributed over time to describe a composition.

Figure 3-4 shows a score fragment consisting of two staves. Each stave contains a treble clef and a number of notes following it. Each note is clearly separated from the preceding note, or clef, by a time delay. The delay following the clef in each case,

[2]The system emphasises the compositional use of notation mentioned earlier in the text.

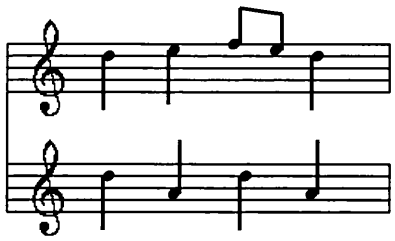

Figure 3-4: *An Example Two Stave Score Fragment*

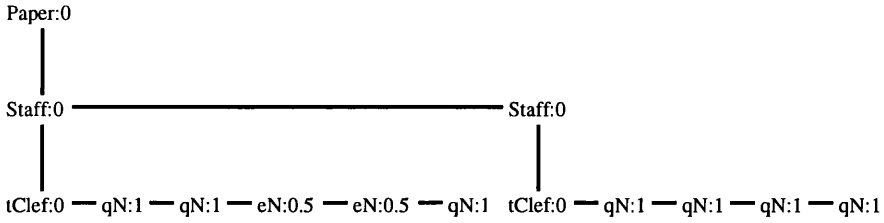

Figure 3-5: *TTree Representation of a Two Stave Score Fragment*

however, is zero. Figure 3-5 shows the score fragment represented as a TTree where *tClef* stands for treble clef, *qN* means quarter note, and *eN* means eighth note, with a whole note being one semibreve. Therefore *qN:1*, for example, represents a quarter note, or crochet, following by a delay of one crochet beat. The object class and delay time can be used either to display each score element in the correct position on the page, or control the order of sound generation events for the purpose of listening to a composition.

The TTree is, therefore, a potential method for constructing composite sounds from smaller elements produced via experimentation. The use of TTrees for this purpose was thus explored by the production of a program for the building and rendering of sounds constructed from smaller elements distributed over time. It was found, however, that the model is suspect to some of the problems found with the DMIX system. Since it is only possible to distribute objects over time, all other aspects of an object, such as the pitch or amplitude of a note, must be changed within the object itself. Hence, as with DMIX, the problem is with the difficulty of undoing alterations made during the compositional process. For this reason the TTree is not completely acceptable as the basis of a system for the composition of experimental computer music. It will be seen in the following text, however, that the new model proposed, which provides a solution to the problems we have described, is in a sense an extension, or generalisation, of the TTree mechanism.

3.4 Building and Manipulating Composite Sounds

In beginning to describe the solution to the problem of building, and manipulating, composite sounds we will first fall back on the analogy between the use of a child's building set, and the paradigm of experimental computer music. When the child builds a house, for example, from the bricks, of various shape and colour, he/she might proceed by building walls, a door, a roof, etc, and construct from these composites, consisting of many joined bricks, the more complex structure of the house. During this process the physical properties of the bricks, which are the primitives in the building kit, remain unaltered. They are, however, transformed in space by both translation and rotation, relative to the structure to which they are attached. This type of building mechanism can be found in the technique known as constructive solid geometry (CSG) [12], found in the subject of computer graphics. In CSG solid objects are constructed via the combination, using set operations such as union, of geometrically defined primitives, which are thus transformable in space.

After the deconstruction of a composite in the child's building set, say the house and thus the doors and walls etc, the bricks are found to be in the same condition as before the construction process. This is regardless of how the house has been "played" or experimented with, by the use of rotation, translation, or joining, which are the methods of experimentation here. This is due to the fact that the bricks only appear different due to their position, and angle, relative to the point at which they are viewed.

3.4.1 Parameter-Spaces

Let us suppose that we have an object, a sine wave, which has a given frequency of 440 Hz, i.e it has a single frequency parameter. Now suppose we wish to raise the pitch of this sine wave by an octave, i.e. raise its pitch to 880 Hz. Suppose that instead of changing the parameter of the wave we consider that our object, the sine wave, occupies a point in a frequency space, and we are simply moving it a distance of 440 Hz within it. The internal slots of the object will remain unaltered. However, since we know its position, when we come to render the sound, we can do so with the correct frequency 880 Hz. In general the values supplied to the pressure function of a sound generating object are determined by its position in its parameter-space. This, as will become clear, solves the problem stated above concerned with manipulating composite sounds, since the objects themselves are not altered, but produce a modified sonic output due to their relative position to the point at which they are viewed. Before continuing on this path we will first stop to examine the nature of these parameter spaces, and in particular their component dimensions.

3.4.2 Dimensions

Notionally we can say that a dimension will consist of a number of points. Each point is a measure of distance from itself to a fixed origin. For example the value 440 Hz can be seen to be 440 Hz from 0 Hz, the origin. We can also move from one point in the dimension to another. Hence there are routes between the points. These routes are relative measures of distance between the points. Given this notional definition we can make the following formal definition.

Definition

We define a dimension in terms of sets Γ and Δ. Then $\forall\gamma \in \Gamma$, γ is an absolute measure in the dimension and $\forall\delta \in \Delta$, δ is a relative measure in the dimension. Finally we have a function Φ of the form $\Phi : \Gamma \times \Delta \longrightarrow \Gamma$. Hence a dimension is defined as a triple $\Omega = (\Gamma, \Delta, \Phi)$. Put simply Γ is a set of points. Δ is a set of routes between the points, and the function Φ is a method for going along a route from one point to another. To aid the understanding of this definition we will give the following two examples.

Examples

1. Let $\Gamma = \Re^+$ and $\Delta = \Re$. We have the interpretation that $\gamma \in \Gamma \Rightarrow \gamma$ is a frequency in Hz and $\delta \in \Delta \Rightarrow \delta$ is a frequency difference in Hz. We therefore have the very simple function $\Phi : (\gamma, \delta) \longrightarrow \gamma + \delta$,
   $\forall\gamma \in \Re^+, \forall\delta \in \Re$ where $\gamma + \delta > 0$. This gives the frequency reached by the displacement δ from γ. Hence $\Omega = (\Gamma, \Delta, \Phi)$ defines one particular dimension of frequency in Hz.

2. Let $\Gamma = \{violin, viola, cello, contrabasso, flute, oboe, clarinet, bassoon, trumpet, horn, trombone, tuba, soprano, alto, tenor, bass\}$. Let $\Delta = \{string, woodwind, brass, voice\}$. We have the interpretation that $\gamma \in \Gamma \Rightarrow \gamma$ is an instrumental timbre and $\delta \in \Delta \Rightarrow \delta$ is an instrument family. The action of the function Φ in this case is when given a timbre γ, and a family δ, the result is the timbre that is the counterpart of γ in δ. That is the instrument in the family that has a similar range. For example $\Phi(viola, voice) = alto$ and $\Phi(flute, string) = violin$. Hence the triple $\Omega = (\Gamma, \Delta, \Phi)$ defines a dimension of timbre. Figure 3-6 shows a representation of this dimension in which each instrument is a node, and the routes between them are given as arcs. For the purposes of clarity in the diagram, only the *string* and *wind* routes are given.

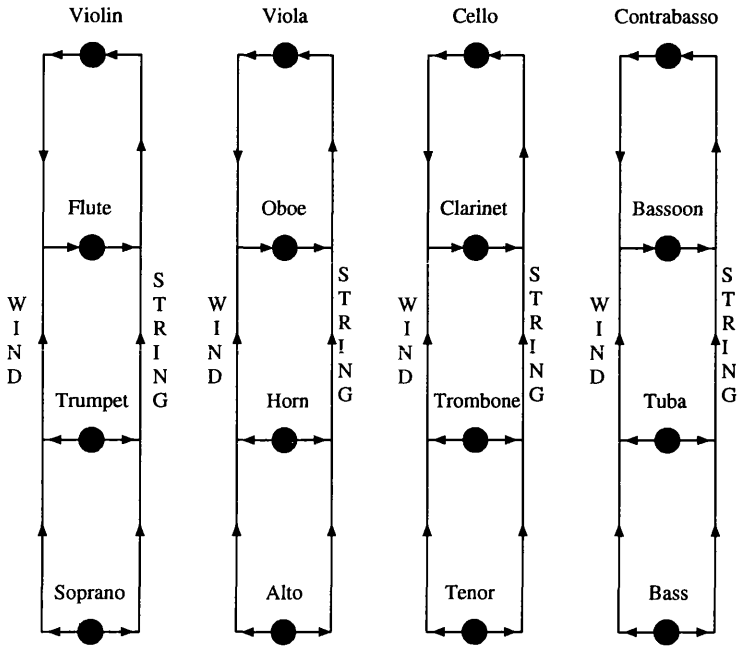

Figure 3-6: *Graph of a Discrete Dimension of Instrumental Timbres*

Given this formal definition of dimension, and hence of a parameter-space, we will now consider the nature of a composite sound constructed in such a space.

3.4.3 Defining the Structure

A composite sound will consist of a tree of nodes to which a number of sound generating objects, e.g. instruments, will be attached. The attachment takes the form of a position in the space of parameters of a particular sound generator, relative to the node. As an example suppose that we have an instrument, with a fixed timbre, which has a frequency value of f, and amplitude a. If we assume a particular ordering of these values we can say that this instrument is at (f, a) relative to the node to which it is attached.

The nodes themselves are connected via the arcs which control the relative position, over time, of one node in relation to another. The motion of a node, relative to another, has an affect on the absolute position in the parameter-space of any sound generating objects attached to it. The values used to drive an instrument, i.e. its parameters, are defined by the instrument's position in parameter-space, relative to an observer. Hence the sound of an instrument perceived at the position of the observer, is affected by the motion of the structure of nodes.

Before we make a definition of the elements of our structure we will first state some

assumptions necessary for their definition. Let us suppose that all the parameters taken by sound generating objects in our system are taken from dimensions contained in a set of dimensions D. In addition to this we also have a dimension of time which we will call T. Having made these assumptions, we may now proceed with our definitions.

Sound Generator A sound generator is a pair $s = (f, p)$, where f is a function of the form $f : T \times d_{s_1} \times \ldots \times d_{s_k} \longrightarrow sound\ pressure$, and where $d_{s_i} \in D\ \forall i = 1 \ldots k$, for some natural number k. In addition p is a tuple of the form $p = (p_1, \ldots, p_k)$, with each $p_i \in d_{s_i}\ \forall i = 1 \ldots k$. Thus a sound generator is essentially an instrument coupled with a position in its parameter-space, relative to the node to which it is attached.

Node A node is a triple $n = (m, a, S)$, where m is the node that is directly above n in the hierarchy, which we will therefore call the *parent* of n. Should there exist no node that is the parent of n, we shall write e in its place. In addition a is a tuple of arcs, the definition for which is found below, of the form $(a_1, \ldots, a_k)$, for some natural number k. The set S is the set of sound generators associated with this node. Again, if there are no arcs issuing from the node we shall write the symbol e in place of the tuple.

In order that the structure of nodes is constrained to be a tree, we assert the following conditions. We have a function $higher : Nodes \times Nodes \longrightarrow \{true, false\}$, which is therefore a predicate function given by the following.

$$higher(n, m) = \begin{cases} false & \text{if } m \text{ is the root node} \\ true & \text{if } n \text{ is the parent of } m \\ higher(n, parent(m)) & \text{otherwise} \end{cases}$$

The predicate function $higher$ evaluates as $higher(n, m) = true$, therefore, if n is higher in the structure of nodes than m. Given this definition we will constrain the structure such that $\forall n \in Nodes, higher(n, n) = false$.

Arc An arc is a triple $a = (n, m, g)$, where n and m are the nodes connected by the arc, such that n is the parent node of m. g is a function of the form $g : D \cup \{T\} \longrightarrow F$, where F $= \{f_d : \Gamma_d \times T \longrightarrow \Gamma_d, \forall\ dimensions\ d = (\Gamma_d, \Delta_d, \Phi_d) \in D\} \cup \{f_t : T \longrightarrow T\}$, such that $g(d) = f_d$, and $g(T) = f_t$. The set F is, therefore, the set of all functions defining the relative motion of the node beneath it in all dimensions from which the parameters of the sound generating objects are taken. The function g simply produces the correct function for the dimension given by its single argument.

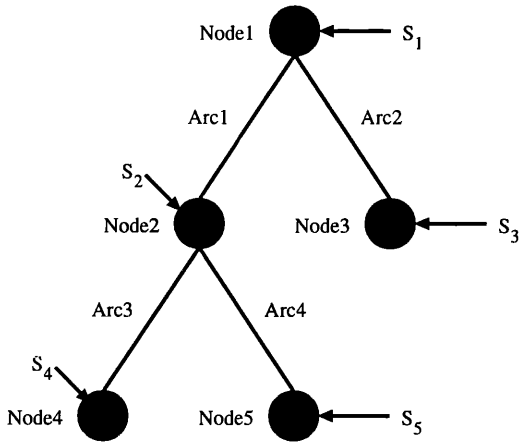

Figure 3-7: *An Example of a Tree of Nodes and Arcs*

Node	*Parent*	*Arcs*	*Generators*
Node1	e	$(Arc1, Arc2)$	$\{s_1\}$
Node2	Node1	$(Arc3, Arc4)$	$\{s_2\}$
Node3	Node1	e	$\{s_3\}$
Node4	Node2	e	$\{s_4\}$
Node5	Node2	e	$\{s_5\}$

Figure 3-8: *Table of Nodes*

3.4.4 An Example Structure

Figure 3-7 shows a tree of five nodes connected by four arcs. Each node has associated with it a single sound generating object, each of which has parameters from a number of dimensions. For example, the sound generating object $s_1 = (f_1, p_1)$, where f_1 is a function of the form $f_1 : T \times D_1 \times \ldots \times D_k \longrightarrow$ *sound pressure*, with each D_i a dimension. In addition $p_1 = (d_1, \ldots, d_k)$, with each $d_i \in D_i$.

The table in figure 3-8 shows the associated values for each node in figure 3-7. Figure 3-9 gives the values for each of the four arcs found in the example tree of nodes.

3.4.5 Parameter Evaluation

The actual parameter values of a sound generator, that is the values which determine the sound that is output, are themselves determined by a generator's position in parameter-space. It is therefore necessary to be able to calculate these values, i.e. the position, when computing the sound for a particular structure of nodes, arcs, and generators. That is when we wish to produce the sound file for a composite sound.

Arc	*Parent*	*Child*	*Function*
Arc1	Node1	Node2	F_1
Arc2	Node1	Node3	F_2
Arc3	Node2	Node4	F_3
Arc4	Node2	Node5	F_4

Figure 3-9: *Table of Arcs*

Before a formal definition of the method of calculation is given we will make the following definitions. u is a function of the form $u : Nodes \longrightarrow Nodes$ such that $u(n)$ gives the node directly above n in the tree, i.e. the parent of n. In figure 3-10, for example $u(Node3) = Node2$. v is a function of the form $v : Nodes \longrightarrow Arcs$ such that $v(n)$ gives the arc directly above n. In figure 3-10 for example $v(Node3) = Arc2$. Finally f_i^j denotes the function for dimension Ω_i of arc j. The time at the node at the top of the tree we will call t_0. When the sound is computed t_0 increases uniformly. The time at node n is given by the function $e_t : Nodes \longrightarrow T$, evaluated thus.

$$e_t(n) = \begin{cases} t_0 & \text{if } n \text{ is the root} \\ f_t^{v(n)}(e_t(u(n))) & \text{otherwise} \end{cases}$$

Hence in figure 3-10 we have $e_t(Node2) = t_1 = f_t^1(t_0)$ and $e_t(Node3) = t_2 = f_t^2(t_1) = f_t^2(f_t^1(t_0))$. Now if n is a node let n_t denote the time at that node. Suppose that an object at a node n has a parameter $p \in \Gamma_i$ for some i. The value during rendering is given by the function $e_i : \Gamma_i \times Nodes \longrightarrow \Gamma_i$ thus.

$$e_i(p, n) = \begin{cases} p & \text{if } n \text{ is the root} \\ e_i(f_i^{v(n)}(p, (u(n))_t), u(n)) & \text{otherwise} \end{cases}$$

Where $(u(n))_t$ denotes the time at the parent node of n. Hence in figure 3-10 the value of parameter p of object A is $e_d(p, Node3) = f_d^1(f_d^2(p, t_1), t_0)$. Therefore if the object A has a pressure function $g : \Gamma_d \times T \longrightarrow$ *sound pressure* the result during performance, or computation, will be $g(e_d(p, Node3), t_0)$, since $e_d(p, Node3)$ is a score for the sound generating object A. The pressure value of the composite at time t_0 is thus given by the summed pressure values of all the objects in the tree.

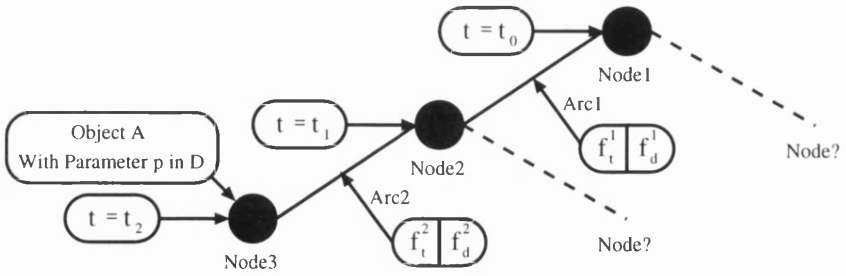

Figure 3-10: *Evaluating the Value of Parameter p of Object A*

3.5 The Rendering of Composite Sounds

3.5.1 Introduction

The term rendering has become almost exclusively associated with the field of computer graphics, in which a model of a scene is rendered to produce an image. This stems from one particular meaning of the word, which is that to render is to translate one thing into another. Hence in the field of computer sound synthesis the term rendering refers to the process of translating a particular model of sound into another representation, and in particular, as in this instance, the low-level digital representation, stored in a sound file for example. This process of sound rendering has also been likened to that of compilation, found in the implementation of many computer languages.

The rendering of the composite sounds defined above consists of four stages. To begin we must generate what we will refer to as the contributor set. This is a set of pairs of the form (s, n), where s is a sound generator which contributes to the overall sound of the composite, and n is the node to which it is attached. Then for each element of this set we must produce the path of the generator in the pair. For each sound generator we will then produce a representation of its path through its parameter-space. This we can then use to provide the parameters for the sound generator over time, to produce the contributed sound of the generator to the entire sound of the composite. The sound for the composite sound is therefore given by the sum of the sound contributed by all the sound generators in the contributor set.

Before we begin rendering the composite sound we must first decide upon the values of a number of parameters. Firstly we must select the node we are to consider to be the root of the area of the composite we are to render. The ability to produce the sound for a particular portion of our composite is important in the context of experimental computer music composition. It is rather like designing the motion of a skeleton, during which we might wish to examine the motion of a particular limb or bone say, minus the motion of it due to its attachment to the body.

After deciding which node to take as root, we must specify an interval of time over which the time at the root node is to run, and hence what interval of time we are to render the sound over. Along with the definition of this interval we must also specify by what step of time the time at the root is to increase, or decrease, as there is, it should be noted, nothing wrong with time running backwards over the interval. This would produce a time reversed output, in the sense that the parameter paths, which will examine later, will be time reversed. The time step we choose takes the role of the control rate, as it specifies the rate at which the parameters of the sound generators are updated, and hence the accuracy to which the parameter paths are represented[3]. Finally we must specify the sampling rate at which the final output is to be produced.

Hence prior to rendering of the composite we have defined the following parameters.

r The node that is to be considered as the root of the portion of the composite we are rendering.

t_1 & t_2 Forming the interval $[t_1, t_2]$ over which we are to render.

k The control rate specifying the accuracy of path rendering.

s The sampling rate of the final outputted sound file.

3.5.2 Producing the Contributor Set

For convenience in the following expression we will assign the functions p, a, and g to be equivalent to the projection functions Π_1[4], Π_2, and Π_3, respectively such that for a given node n, $p(n)$ gives the parent of n, $a(n)$ gives the arcs of n, and $g(n)$ gives the sound generators attached to the node. In addition, for a given arc r the function $c(r)$ gives the node to which the arc points, i.e. the child of the arc. We further specify a function called *size*, which gives the number of elements in a particular n-tuple. Hence for example $size((\alpha_1, \alpha_2, \alpha_3)) = 3$. We also require the function *pairs*, which gives the required set of pairs of node and generator, given a the set of generators for the node. For example, if $s(n) = \{s_1, s_2\}$ then $pairs(n) = \{(n, s_1), (n, s_2)\}$. Now if we let P represent the set of all pairs of nodes and sound generators, the contributor set is given by a function of the form $Con : Nodes \longrightarrow P$, evaluated as follows.

$$Con(n) = \begin{cases} pairs(n) & \text{if } a(n) = e \\ (\bigcup_{i=1}^{size(a(n))} Con(c(\Pi_i(a(n))))) \cup pairs(n) & \text{otherwise} \end{cases}$$

[3]The nature, and production, of parameter paths will be discussed later in the text. In the section on parameter path rendering we will examine the issue of accuracy

[4]For an n-tuple m, the function $\Pi_i(m)$ gives the i^{th} element of m, $\forall i = 1 \ldots n$.

Therefore in our example structure in figure 3-7, given *Node1* as the root node the *Con* function will yield the following result.

$$Con(Node1) = \{(Node1, s_1), (Node2, s_2), (Node3, s_3), (Node4, s_4), (Node5, s_5)\}$$

Giving the remaining nodes as the root of our structure, we get the following results.

$$Con(Node2) = \{(Node2, s_2), (Node4, s_4), (Node5, s_5)\}$$

$$Con(Node3) = \{(Node3, s_3)\}$$

$$Con(Node4) = \{(Node4, s_4)\}$$

$$Con(Node5) = \{(Node5, s_5)\}$$

3.5.3 Generating the Parameter Paths

Once all the sound generators, which contribute to the composite, have been collected, the next stage is to create the paths of the parameters for each generator. A parameter path is a history of the value of a particular generator parameter over the time for which we are rendering the composite. As such it is essentially a function of time which yields a parameter value as output. For example, let us consider that we have a generator which produces sound pressure output given a value of time, and of frequency. Let us further suppose that the frequency value is taken from a dimension $F = (\Gamma_f, \Delta_f, \Phi_f)$, and that we are rendering over the interval of time $[t_1, t_2]$. Then the path for the frequency parameter is represented by the function $p : T \longrightarrow \Gamma_f$. Hence the task of generating the path for this parameter is equivalent to the task of constructing a representation of this function.

When this is achieved for all the parameters of a particular generator, we have in effect constructed a score, in the sense we defined previously, for the generator in question. To demonstrate this fact suppose that we have a sound generator g with a sound generating function $f_g : T \times \Gamma_1 \times \Gamma_2 \longrightarrow$ *sound pressure*, that is g has two parameters, in addition to the time parameter. Hence if we now generate the paths for these parameters over the interval $[t_1, t_2]$, and call these p_1 and p_2 respectively, then the pair (p_1, p_2) is clearly a score for g. This is because the evaluation of $f_g(t, p_1(t), p_2(t))$ where t goes from t_1 to t_2, produces the contributed sound for this generator, which is in effect an instrument in all but name, for the portion of the composite we are interested in here.

What we have to solve now is the problem of generating the paths for all the generators in the contributor set. Hence we must generate the parameter path for each

generator's parameters. Therefore to generate a particular parameter path we are given the interval $[t_1, t_2]$, the control rate k (specified in Hz), the initial parameter value p, and the node n, in which the generator, to which the parameter belongs, is found. Let us suppose that our parameter $p \in \Gamma_p$, and hence the path of p is a function of the form $f : [t_1, t_2] \longrightarrow \Gamma_p$. We have already seen how to evaluate the value of a parameter at a given node. This was given by the function $e_i(p, n)$, where i was an index to the dimension from which the parameter p was taken. Therefore the path is given by $f(t) = e_{\Gamma_p}(p, n)$ given that in the equation for e, $u(n)_t = e_t(u(n))$,[5] with t_0, the time at the root node, given by the current value of time at the root node $t \in [t_1, t_2]$.

The above solution for the path of the parameter p is exact for all values of t in the interval $[t_1, t_2]$. However, for the purposes of rendering we are only interested in evaluating the path at points along the interval $[t_1, t_2]$ separated by the amount specified by the control rate k, i.e. at steps of $\frac{1}{k}$. Hence we evaluate, as above, $f(t)$ for $t = t_1 + \frac{\alpha}{k}$ for a non-negative integer $\alpha = 0 \ldots \delta$, with $\delta \leq k(t_2 - t_1) < \delta + 1$.

The question arises as to how we are to give values for f where t does not lie on one of the chosen points. This decision is fairly arbitrary, though the result will likely have a bearing upon how much aliasing[6] will occur in the path rendering process. Two possible schemes are outlined below. The first chooses to take the value of the nearest evaluated point that is less than t. The second views the path as being constructed from straight lines joining the evaluated points. This straight line interpolation is likely to be more accurate than the first representation. It is likely, however, that there will exist better methods of representation than than this. For example, the fitting of a curve to the given set of points may, in many circumstances, produce better results.

1. $f(t) = f(t_1 + \frac{\alpha}{k})$ where α is a non-negative integer, such that $\alpha \leq k(t - t_1) < \alpha + 1$.
2. $f(t) = f(t_1 + \frac{\alpha}{k}) + \frac{t - t_1}{t_2 - t_1}(f(t_1 + \frac{\alpha+1}{k}) - f(t_1 + \frac{\alpha}{k}))$, where α is a non-negative integer, such that $\alpha \leq k(t - t_1) < \alpha + 1$.

It should be noted that the arithmetic operations in the equations, are generic, with the action required defined in terms of the dimensions to which the operands belong. To illustrate the process of path rendering we will, in the following text, give a brief example.

[5] u(n) gives the parent of the node n.

[6] By the term aliasing we mean differences between the continuous representation of the path, and its representation using points sampled at the control rate. Aliasing can occur in any system which represents a continuous process as a number of values sampled at particular intervals in time, for example the digital representation of sound. The greater the intervals at which sampling takes place, i.e. the lower the sampling rate, the greater the risk of aliasing.

An Example of Path Rendering

As an example, let us consider that we are rendering a structure as found in figure 3-11. Here we have a sound generator, a fixed amplitude sine wave generator to be exact, at the bottom of our hierarchy of nodes. The only parameter that this generator takes is one of frequency in Hertz, which is currently 440 Hz, say. The arcs above the node, in which the generator is found, contain frequency functions f_2, and f_1 respectively. For convenience we will assume that all time functions are identity functions, and hence the time will be the same for all nodes, namely the time at the root node, here Node1. We are to be rendering this structure with Node1 as the root, over the interval of time $[0, 4]$ seconds, and at a control rate of 10 Hz. We do not need to concern ourselves with the sampling rate, as we are only concerned with the production of paths here. The function f_1 is defined as $f_1(p, t) = 100t + p$, and the definition of f_2 is given below.

$$f_2(p,t) = \begin{cases} 100 + p & \text{if } t < 1 \\ 200 + p & \text{if } t < 2 \\ 300 + p & \text{if } t < 3 \\ 300 - 200(t-3) + p & \text{otherwise} \end{cases}$$

A graph representing f_1 with p fixed at zero can been seen in figure 3-12. A similar graph for f_2 can be seen in figure 3-13. With the use of the equations for time and parameter values shown earlier in the text, we can calculate the path of the frequency parameter. If we were to use the system of giving parameter values for times in between those that are rendered as being the value of the nearest rendered value before it in time, we get the path shown in figure 3-14. This path may now be used to drive the sound generator to produce the sound for its part of the structure, which in this example is the sound for the whole structure.

3.5.4 Dimension Hierarchy

Before dealing with the final production of the pressure function for our composite we will examine an small issue relating to the control of parameter values. It may often be the case that when computing parameter values we wish particular values not to be affected, whilst others of a similar type must change. This occurs when we have vibrato, given as a frequency which we do not wish to change, and note frequencies which we wish to be altered. This is not a problem since we can view the dimensions of vibrato frequency and note frequency, i.e. pitch, as different entities which are therefore governed by different arc functions. In a sense we have a kind of class hierarchy as shown in figure 3-15. Here the class of frequency parameters is a subclass of the class

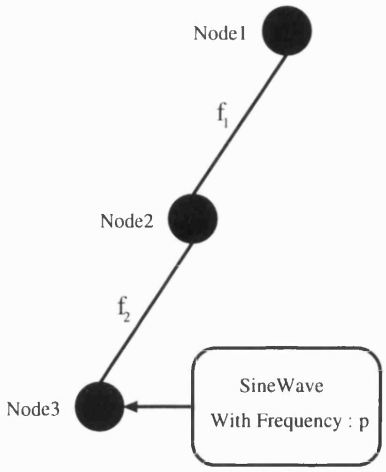

Figure 3-11: *An Example Node Tree Structure to Illustrate Path Rendering*

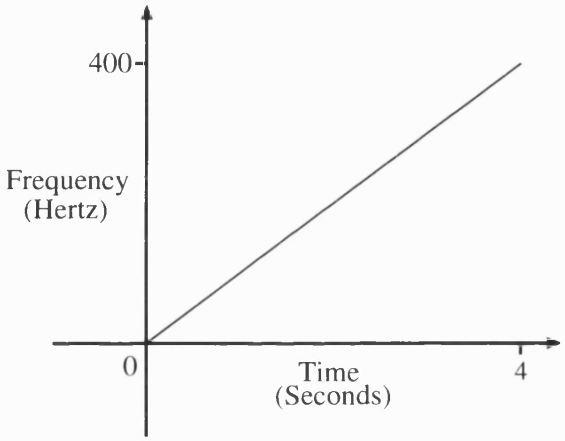

Figure 3-12: *Graph of Frequency Function* f_1

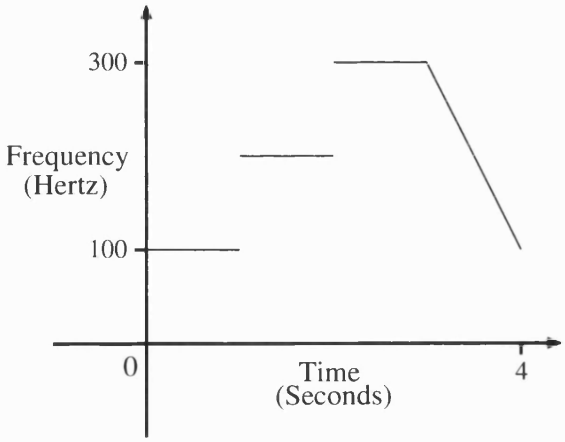

Figure 3-13: *Graph of Frequency Function* f_2

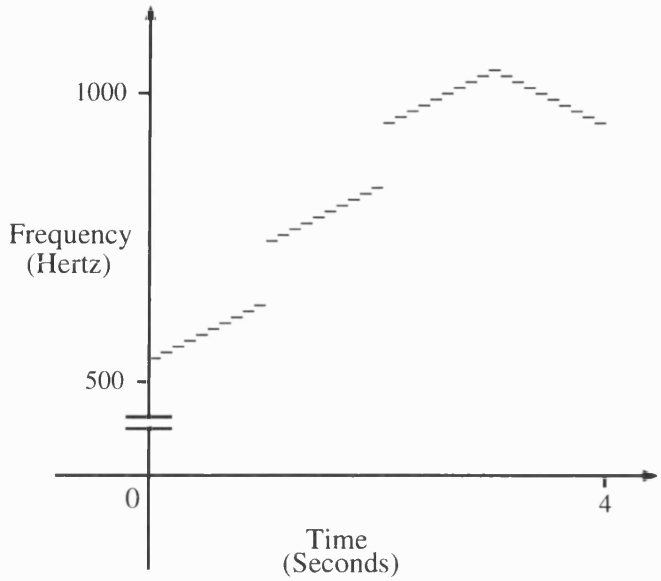

Figure 3-14: *The Path of the Frequency Parameter as Rendered.*

of all parameters. The frequency class is further divided into vibrato and pitch classes. Hence each of these properties can be controlled separately by defining a function for its class, or together if desired, by defining a function for their superclass.

3.5.5 Production of The Pressure Function

The production of the pressure function, i.e. the sound, for the composite is a simple process of using the paths for each sound generating object in the contributor set, to drive the associated sound generator. Hence the path for each sound generator is used as a score. The resulting sound from each sound generator's performance of its path, is

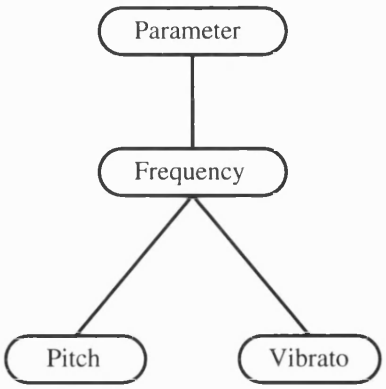

Figure 3-15: *Dimension Class Hierarchy*

summed to produce the sound for the entire composite. For example, suppose we have a number of sound generating objects $s_1 \ldots s_k$, for some natural number k, with each $s_i = (f_i, p_i)$. Each of these generators takes a single parameter of frequency, and for each of these parameters we have produced paths $p_1 \ldots p_k$. We further suppose that the composite is rendered over an interval of time $[0, d]$ seconds, and that the sound for the composite is given by the function $g : [0, d] \longrightarrow$ *sound pressure.* The function g is, therefore, evaluated by the expression $g(t) = \sum_{i=1}^{k} f_i(p_i(t))$.

3.6 Summary of the Nodes and Arcs Model

The model of a composite sound as a tree of nodes, to which are attached sound generators, connecting by arcs moving in parameter-space can be seen to be a generalisation of data structures for music such as the TTree [6], of Glendon Diener. This however only controls the position of musical objects in time, and hence forms a method of starting and stopping hierarchies of sound generators. The nodes and arcs model generalises this by providing a method of constructing sounds in any musical space, which we have called parameter-spaces. This not only provides a method for applying well known effects such as tempo curves [5] to provide performance nuances, but also more general and abstract notions such as being able to apply a glissando to any sound regardless of the complexity of its construction, or transforming a string quartet, as a single unit, into a wind quartet.

The purpose of the design of the nodes and arcs model was to form a solution to the problem of score specification, in an experimental computer music system. This was found to be a problem of providing a mechanism of constructing composite sounds hierarchically, which could then be experimented with, without affecting the components from which the composite is built. This is achieved by the model since at no time are the sound generating objects altered internally. The sonic effects are produced by their motion in their parameter-spaces, relative to a particular observer, i.e. the root node at rendering.

The next problem is how to represent this process in an uncomplicated manner to the user of an experimental computer music system. It is therefore the problem of designing a suitable user interface for a system which uses the nodes and arcs model. The solution to this problem, used in the implementation of a prototypical system, will be described in the following chapter. This will be described within the context of a prototypical system which has been implemented, and tested, which forms the overall topic of discussion in the following text.

Chapter 4

Implementation of a Prototypical System

4.1 Introduction

The purpose of the implementation of a prototypical system was to provide a platform from which to test the viability of the nodes and arcs model as a solution to the problem of score specification under the paradigm of experimental computer music. The system is, therefore, not intended to be a complete and general package for sound synthesis. It is rather an implementation of the aspects of a system which uses the nodes and arcs model, which are new. We will begin the description of the system by first examining the system architecture.

4.2 System Architecture

Figure 4-1 shows the architecture of the prototypical system. The main components, as can be seen from the diagram, are the *Graphical User Interface* (GUI), *Lisp Object System*, *Sound Renderer*, and *Audio I/O*, i.e. audio input and output. The fifth module, *Disk*, simply represents backing store for the input and output of sound files, as such it is not strictly a part of the system. The audio input and output module is of no real interest, and as it simply consists of program code for the audio input and output of sound data already existing either in memory or on disk.

The main components of this system, which implement the preponderance of its functionality, are therefore the GUI, object system, and sound renderer. Of these three modules the object system is the only one for which its implementation was not platform dependent. The host platform for this implementation was a Silicon Graphics

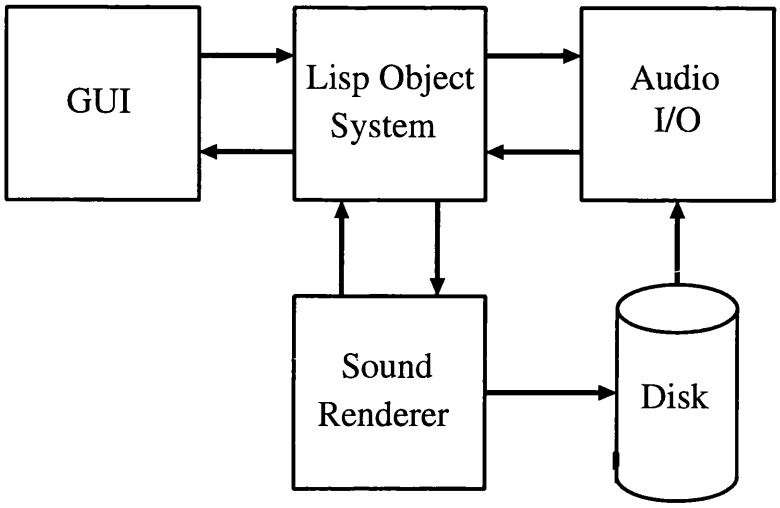

Figure 4-1: *System Architecture.*

Indigo, with excellent graphical and audio[1] capabilities.

4.3 Graphical User Interface

4.3.1 Design

The production of a graphical user interface to a particular process is in essence the realisation of a particular metaphor for the process in question. We have already, in the preceding text, referred to the similarity between the paradigm of experimental computer music, and the action of play, and in particular play and experimentation with a building kit. Whatever user interface design we are to choose, it must represent the objects found in our model, namely nodes, arcs, functions, scores and tools on screen, and provide for the interaction of these, via the action of the user. There are, perhaps, a large number of metaphors which we might use to represent the process of building sounds using the nodes and arcs model[2], all of which may be perfectly usable. Indeed it might be argued that the most useful interface would be one which allows for the design of a particular metaphor that each particular user finds most appropriate. We are not, however, interested in the design of a user interface which demonstrates some new technique in human computer interaction (HCI), or attempting to provide a GUI from the analysis of the need of composers [49]. We simply wish to have a usable interface, which provides a means of testing the nodes and arcs model. The selection of a metaphor by which to represent the compositional process using the model is,

[1]Supports stereo sound at maximum sampling rate of 48 kHz, with 16 bit samples
[2]See section on Future Research : User Interface Issues

therefore, an arbitrary one. At the very least a particular solution to the user interface problem will not be unique in its success. In spite of this, we will give a brief discussion on the selection of the metaphor, which is implemented in the prototypical system.

In the nodes and arcs model there are a number of classes of items, or objects, that are used in the compositional process, which require a representation in the user interface. First and foremost, there are the nodes and arcs themselves. The GUI represents the structures built from these objects as rooms (the nodes), connected to each other via doors (the arcs). All objects attached to a particular node are seen to inhabit the room which is the representation of the node in the GUI. The doors, which inhabit the room from which the arcs they represent stem, act like arcs by controlling the relative position of the nodes, and hence rooms, below them in the hierarchy. In a sense what happens when rendering is that the sound in a particular room is the result of sound passing through the doors from rooms lower in the hierarchy, with the addition of sound generated in the room itself. In addition the action of each door warps the sound coming through it in a particular fashion, which is defined by the user's application of functions to it. This models the functionality of an arc.

In order that the user can move between many rooms, at different positions in the compositional hierarchy, describing different parts of the composition, it should be possible to view many parts of the structure at once, and move between them. The view of a room is, therefore, that of a window showing a movable viewport onto the room in question. With the production of a number of these viewports, showing a multiplicity of rooms, the user is able to move between them, using the mouse, and focus his/her attention on a particular part of the composition at will. Since we wish to show a number of viewports on screen simultaneously, we would like to minimize the size of the windows they inhabit. For this reason the objects found in each room are displayed in iconic form on a grid, shown by the viewports, mimicking the object matrix found in the internal representation of nodes, which we will examine later in the text.

To enforce the sense given to the user of actually working inside the compositional structure, rather than externally creating it, it was decided that the system's data, and tools, for object creation, destruction, and manipulation etc, should inhabit rooms also. These objects, which do not generate sound, are therefore viewed in iconic form, as part of the rooms they inhabit. This uniform representation of all our system objects generates the problem of how we are to provide interaction between them. To do this we borrow from the paradigm of *slappability* [43], introduced by Daniel V. Oppenheim as part of his system for experimental music composition, DMIX. Actions are therefore produced by the grabbing of an iconic representation of an object from a viewport, and

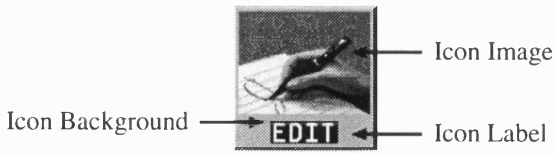

Figure 4-2: *Icon Structure*

the dropping it on any object shown in any viewport. The action taken depends upon the classes of the two objects involved. Hence a single binary generic function is used as the control for the entire system. This function we call the *perform* function since it provides for a kind of abstract form of performance. In the case of an instrument object being dropped onto a score, we might expect to hear the instrument play the score in question. This is a fairly traditional form of performance. We might expect, however, that an object called edit, when dropped upon a score would produce an appropriate editor for the type of score used. In a sense the edit object *performs* the score, by examining its class. The result of the performance is the production of the appropriate score editor.

At this point we have constructed a complete metaphor for the process of experimental music composition using the new nodes and arcs model. The GUI will view nodes as rooms in which all the classes of system objects can be found, and picked up and dropped to invoke the *perform* function, and hence produce the particular action that is desired. We will now examine the implementation of this metaphor in the prototypical system, and in addition detail some of the tools, and data, objects available in the system when it is started, and the functionality that they provide.

4.3.2 The View Library

The view library VL, which was built by the author of this thesis using Silicon Graphics library GL [32, 54] and the user interface library FORMS [44], is a C library which consists of functions for creating, destroying, and manipulating viewports and the icons that inhabit the areas that the viewports inspect. Each viewport, on its creation, is bound to an empty two dimensional array, of a given size, into which icons may be placed, or removed.

The structure of the icons in VL is shown in figure 4-2. Each icon consists of a square background image onto which is placed a smaller image, which is used to identify the icon. At the bottom of the icon a text label is placed to further help identify the object which the icon represents.

The structure of the viewports in VL is shown in figure 4-3. The move button allows the viewport to positioned on screen by the user, whilst the resize button permits the

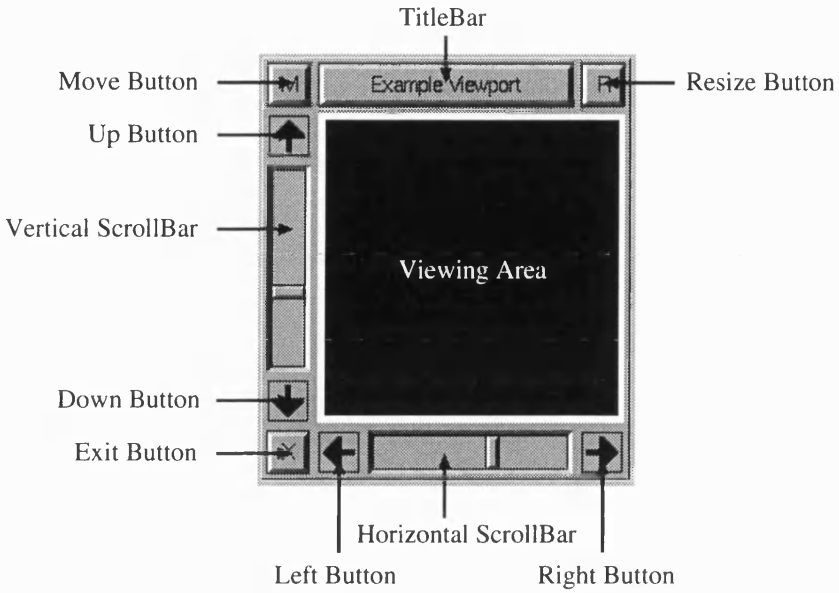

Figure 4-3: *Viewport Structure*

size of the viewing area to be altered. The scroll bars, and arrow buttons are used to control the position of the viewing area over the array of icons that the viewport examines. The exit button is used to close the viewport.

The VL library contains functions which allow the creation and on-screen drawing of viewports. The viewports show a matrix, of a specified size, containing VL icons. The icons themselves may be created and destroyed and added to viewports via the use of additional functions also contained in the library. A function may be called to allow the control of any viewports shown, and the icons within them, in the manner specified above. This function will return only if one of the following two events occurs.

1. A viewport is closed by pressing its exit button.

2. An icon is picked up, by clicking on it, and dropped into a viewport (possibly the same one) with a second click of the middle mouse button.

In the second of these events a library function may be used to obtain handles on the viewport from which the icon was picked up, its position in the icon matrix, the viewport in which it was dropped, the position in the icon matrix of the viewport in which it was dropped, and the icon, if any, that resides at the dropping position. This information can then be used to drive the perform function.

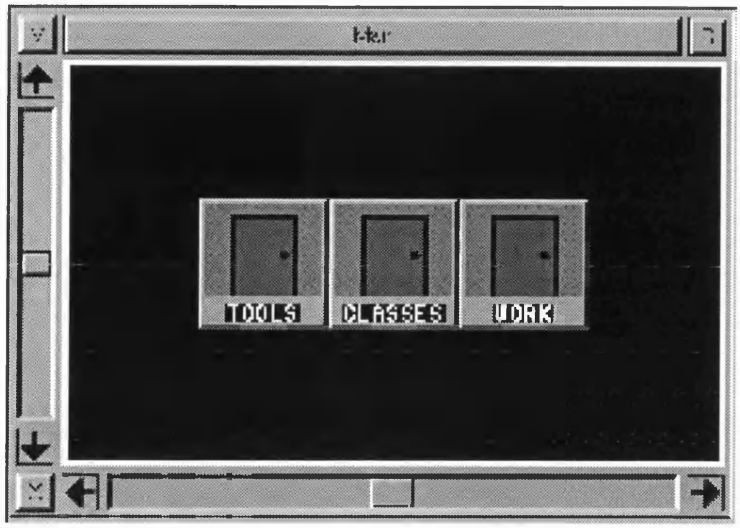

Figure 4-4: *The View of the Main Room on First Running the System*

4.3.3 The Viewing of Rooms

The object system module keeps track of all the objects associated with a particular node in the tree structure. This is in the form of a matrix which can be mapped onto a matrix of icons in order to display the node, or rather room which represents it, in a viewport. This is achieved by first creating a viewport of a size specified by the user, which views an icon matrix, the size of which is identical to the size of the object matrix. A unary generic function is then used to create the appropriate icon for each object in the matrix, which is then placed at an identical position in the viewport's icon matrix. Hence the appearance of a particular object, i.e. the appearance of the icon which represents it in the GUI, is determined by its class and hence the method of the generic function which builds the icon for its class. Finally, the viewport, with its filled in icon matrix, is displayed to allow the user to interact with it in the manner described above. Figure 4-4 shows the appearance of the main, or root room, which is at the top of the initial structure when the system begins. As can be seen, this room, and hence the associated node, contains three doors to rooms lower in the structure. These three initials rooms contain user tools, classes of objects that can be created, and some example note objects.

4.3.4 Invoking the Perform Function

When an action occurs in the user interface the details concerning the pick and drop motion of the icon in question is sent back to the object module. From this information,

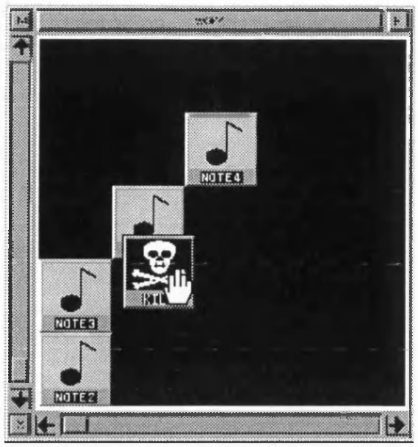

Figure 4-5: *Performing the Kill Tool on a Note to Destroy It.*

and the room to viewport bindings, the object module is able to find out which object has been performed upon which other object, if any. If it is the case that the position onto which the first icon was placed is empty, then the icon, and the object it represents, is moved into this position. Otherwise the perform function is evaluated with the two objects in question as arguments. The object whose icon was moved forms the first argument, and the object on which it was placed becomes the second. After the perform function is evaluated, the objects, and the icons remain unchanged, except in cases such as the deletion of an object with the kill tool. This type of operation is shown in figure 4-5 in which the kill tool is being used to delete a note object from a room. Hence, the perform function has a method which deletes an object when an instance of a kill tool forms its first argument.

4.3.5 Implemented Objects

We will now examine the nature of the objects that are implemented in the prototypical system.

Notes These objects are a very simple form of sound generator with internal slots for frequency, amplitude and timbre. The model for the timbres found in note objects is described later in the section on track production. The note objects in the system are different from MIDI notes as they cannot be "switched" on and off. They will simply play a continuous tone, the properties of which are defined by its motion through its own parameter-space.

Doors Door objects provide the user with a representation of arcs to which functions may be added. Each door also provides a mechanism for displaying the room

which is directly below it in the hierarchy. This is achieved by simply double clicking on a particular door. If the room below the door is not displayed, then a viewport showing the room in question is produced. Rendering is achieved in the system by applying the render tool to a door. The user can then select the parameters, described earlier in the text, which determine how the rendering is to be carried out. After the completion of the on screen form, the sound for the room in which the door resides, or the one directly below it (this choice is made in the form), is rendered and an iconic representation of the sound file produced is given to the user to place in a room. This sound file can then be played by applying the play tool to it.

Tools Tool objects are used to inspect, create, destroy, or modify in some way, the objects that are available in the system. The *edit* tool, for example, may be applied to another object to invoke the appropriate editor for the object in question. In particular the *edit* tool can be applied to a graph object, as described below, with the result that a representation, that can be edited, of the graph object is produced on screen. This representation is in the form of a graph editing tool, as shown in figure 4-6.

Graphs Graph objects provide a general score functionality in that they can be created and edited and applied to doors, to affect the motion in parameter-space of the node, represented by the room, directly below the door in the hierarchy, and hence all sound generators, and other nodes, positioned relatively to it. The correspondence between the graph and the parameter function is defined by the completion of an on screen form, which the user is given to complete, when the graph is applied to a door. Typically values to be specified in the form are which dimension to supply control over, over what time period to do it, and scaling factors.

Classes Class objects exist to represent the classes of all the instantiable objects in the system. For example a class object exists to represent the note class. When the make tool is applied to a note class object a form is produced which the user must complete to specify the initial values of the internal slots of a note object. When this form is completed a note object is created with its internal slots specified as in the completed input form.

GrainGenerators Grain generators produce produce randomly distributed sinusoidal grains (or sonic quanta) over frequency, amplitude, and time. Their operation is therefore governed by parameters describing the probabilistic distribution of

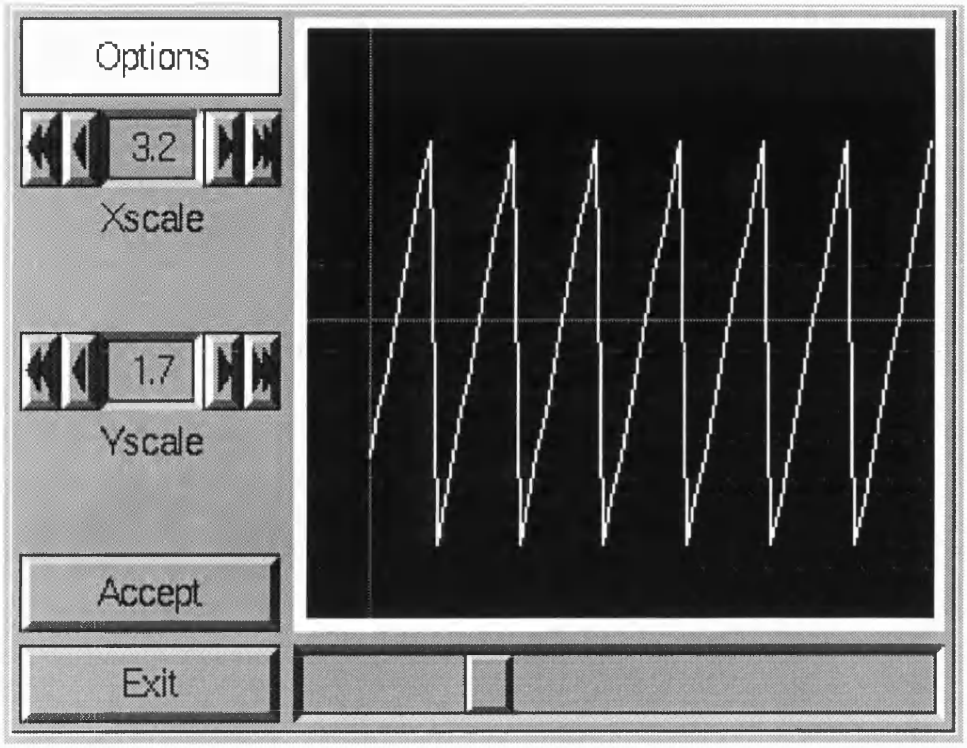

Figure 4-6: *The Graph Editor*

the grains over frequency and amplitude, the average number of grains per second, and the duration of the grains. The grains are assumed to be uniformly distributed over time according to the number of grains per second. Granular synthesis and the use of a grain generator will be described in more detail in the system demonstrations section, given later in the text.

4.4 Object System

The object system is central to the operations of the system, as it acts as the control mechanism for the software. All sound generating objects, tools, the nodes they inhabit, and the arcs that link the nodes, exist in this module. The user interface module represents the objects found in this module in a graphical form. All actions that take place in the GUI are sent to this module, and it is here that a decision is made on what is to be done. The result of a particular action may be to invoke the sound renderer, or the audio input and output module. This process is controlled by the binary generic function *perform*, the arguments to which are specified by GUI action. Hence, the behaviour of the system is defined by the methods of this generic function.

The class hierarchy is important for the purpose of lessoning the number of methods, and is therefore outlined in the following section.

4.4.1 Class Hierarchy

The hierarchy of classes for the object system is shown in figure 4-7. The driving concern in the construction of this hierarchy was the desire to minimize the number of methods that need to be defined for the *perform* function. For example all classes of objects are a sub-class of the *TopLevel* class. This is useful, as will be shown later, as it enables the *perform* function to trap actions for which no specific method is defined.

4.4.2 The Perform Function

All actions in the system, as was stated previously, are controlled through a single binary generic function, the arguments (two objects) to which are determined by the user interface. The Lisp code shown in figure 4-8 shows the declaration of the *perform* function as a generic function of two arguments. This code also defines a method on the perform function for the case of two objects whose classes match the top level class in our hierarchy. This method therefore provides a default action for when no more specialised action is specified. A message is therefore displayed by the user GUI to inform the user that the system does not know how to deal with the performance of these two objects. The language used in this example, and in the implementation of the object module, is EuLisp [45], which was chosen because of its object oriented capabilities, and the properties of Lisp that are ideal for system prototyping.

4.4.3 Structure Representation

The internal representation of the nodes and arcs structure is essentially the representation of a tree of nodes. A particular node object will have slots defining the parent of the node, if one exists, a matrix containing objects that inhabit, or are attached to, the node, and a list of arcs which connect the node to further nodes lower in the hierarchy. Instances of the arc class have a slot containing the node object, lower in the hierarchy, to which the arc points, i.e. the child node of the arc. An arc object will also have a slot which contains a unary generic function, which acts as a function selector when given a parameter of a particular class, i.e. taken from a particular dimension. For example, when given a frequency parameter, the function selector might provide the function by which the position in the dimension of frequency, of the object containing the parameter, is defined relative to the node above the arc. Should there be no particular function defined as the return for a particular class of parameter, an identity

```
TopLevel
    Data
        Classes
            ParentNode
            ChildNode
            GraphScore
            NoteClass
        Doors
        Functions
        GrainGenerators
        Notes
        Parameters
            Amplitudes
                SampleSize
            Frequencies
                Hertz
            Timbres
                WaveTable
            Times
                Seconds
            Stochastics
                Distributions
        SoundFiles
            RenderedSoundFile
            ImportedSoundFile
    Tools
        Copy
        Edit
        Kill
        Play
        Query
        Render
        Make
```

Figure 4-7: *Class Hierarchy*

```
(defgeneric perform-function (x y))

(defmethod perform-function ((x top-level-object)
                             (y top-level-object))
  (vl-show-message "No method is known" "to perform these objects." ""))
```

Figure 4-8: *Code Declaring the Perform Function and Default Method.*

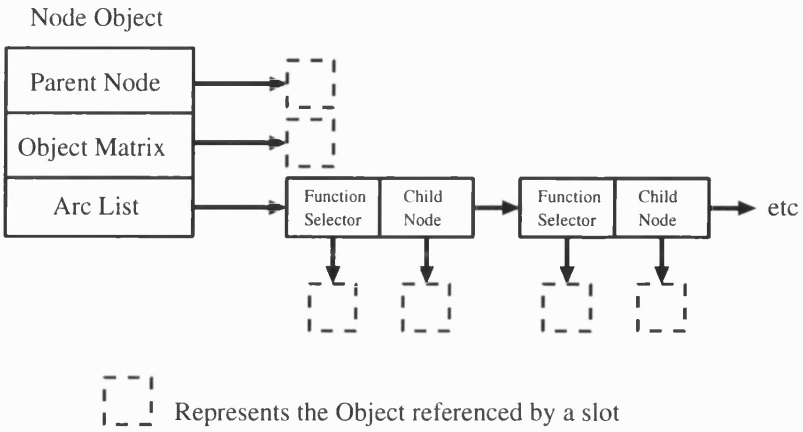

Figure 4-9: *Internal Representation of the Structure of Nodes and Arcs.*

function is returned. Figure 4-9 shows the objects, and their slots, which define the internal representation of a structure of nodes and arcs.

4.5 Sound Renderer

4.5.1 The Contributor Set

The production of the contributor set, the set of sound generators which contribute sound to the total sound of the composite, follows almost exactly the definition given in chapter 3. The only difference is that in the prototypical system not all objects attached to nodes, and hence the rooms that are their graphical representation, are sound generating objects. Hence an object is only added to the contributor set if it yields a positive response from the generic unary predicate function *is-sound-generator*. The Lisp code found in figure 4-10 is an outline of the code found in the system implementation for the generation of the contributor set. The names of the functions and symbols found in the figure, however, are not necessarily those found in the actual code.

The code in figure 4-10 produces a representation of the contributor set by recursing down the structure of rooms, adding the an object found in a particular room if it is a sound generator. When the function *get-contributor-set* is called on the root of the composite sound we wish to render, a list of pairs is produced. Each pair in the list contains a sound generating object and the room, or node, to which it is attached. The resulting list can then be used for the purpose of path rendering, which we will describe in the following section.

```
(defun get-contributor-set (root-room)
  (setq contributor-set ())
  (get-contributor-set-aux root-room))

(defun get-contributor-set-aux (room)
  (add-room-objects room (get-room-objects room))
  (get-contributor-set-exits (get-room-exits room)))

(defun add-room-objects (room objects)
  (cond
   ((null objects))
   ((is-sound-generator (car objects))
    (setq contributor-set (cons (cons room (car objects))
                                contributor-set)))
   (t (add-room-objects room (cdr objects)))))

(defun get-contributor-set-exits (exits)
  (cond
   ((null exits))
   (t (get-contributor-set-aux (car exits))
      (get-contributor-set-exits (cdr exits)))))
```

Figure 4-10: *Sketch of Code to Generate the Contributor Set*

```
(defun evaluate-parameter-value (p room)
  (cond
   ((is-root room) p)
   ((t (let ((p-function (get-parameter-function p room)))
        (evaluate-parameter-value
          (apply-p-function p-function (get-parent-time room) p)
          (get-parent room)))))))
```

Figure 4-11: *Sketch of Code for Parameter Evaluation*

4.5.2 Path Representation and Production

The production of the path for a particular sound generating object, that is a member of the contributor set, is simply a matter of evaluating each of its parameters at each moment in time specified by the interval over which we are rendering, and the control rate. A sketch of the code for the evaluation of a particular parameter value, of an object in a particular room, with the assumption that the time in each room in the structure has been updated, is shown in figure 4-11. The code evaluates the parameter value in a recursive fashion by applying the appropriate function for the class of the parameter, found in the arc above the current room, to the parameter value and the time in the parent room of the room in which we are evaluating. The recursion continues by evaluating the resulting value of the parameter function, in the room that is the parent of the current room. The recursion ends when the root room is reached, in which the parameter value is returned unaltered.

The representation of a particular parameter's path is, therefore, a list. The path for a particular sound generating object is the collection of all its parameter paths, represented as lists. The parameter path lists are not, however, simply lists of parameter values for each control cycle. Rather, they are represented as spanning lists such that if a particular parameter does not move, i.e. change, in ten control cycles, say, the list representing the path will contain the pair (10 . p) where p represents the steady parameter. Figure 4-12 shows the spanning list representation of a frequency path, rendered for four notes each of one second in length, with paths sampled at a rate of 10 Hz. This representation is useful since it allows the track renderer, which we will discuss in the following section, to copy previously written sound samples should the parameters of a sound, which affect the wave that is output, not change. This helps to speed up the process of track rendering, but is only possible because of the simple model of timbre used in the prototypical system, and only applies to sound generators that produce the same sound given the same parameters. This is not the case, for

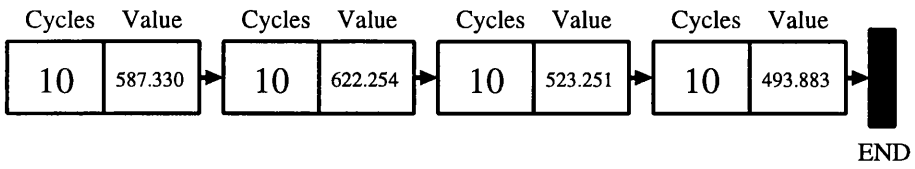

Figure 4-12: *A Parameter Path Represented as a Spanning List*

instance, with a stochastic grain generator.

4.5.3 Timbral Model and Track Rendering

Whilst instrument specification is an important problem in the field of computer sound synthesis, and something that should be dealt with effectively in any useful sound synthesis system, it is not the focus of the research of the author with relation to this thesis. The single timbral model, and hence method of instrument specification, used in the prototypical system is a very simple one. Each timbre in this model consists of a single sound sample of a known frequency, amplitude, and sampling rate. Transformations in pitch are achieved by "stretching" or "compressing" the sample, by the correct amount to produce a sample at the desired frequency. Amplitude modifications are simply achieved by scaling the individual sample values in the sample itself. In addition to these properties, the timbre model allows for the specification of a repeat position. This is the individual sample number, in the list of samples making up the whole sample, at which point the sound sample will begin to repeat after the end of the sample has been reached. An example of a timbre of this type is shown in figure 4-13.

Instances of this class of timbres form part of the internal structure of all the sound generators, such as note objects, found in the system, but note the grain generator objects which only produce sinusoidal grains. Hence all sound generators in a particular contributor set will have timbres of this nature. The rendering of a track, the sound for a single sound generator which contributes to the sound of the composite over the period of rendering, is produced by using the path for the sound generator to drive the modifications to the outputted wave, which is continuously written to a sample store file for the length of the interval of rendering. As in theory, the path for the sound generating object is in effect used as a score for it to perform. Hence for each sound generating object a track is written which represents its contribution to the composite. The track, and therefore sound, for the composite is produced by adding the tracks produced by each of the component sound generators.

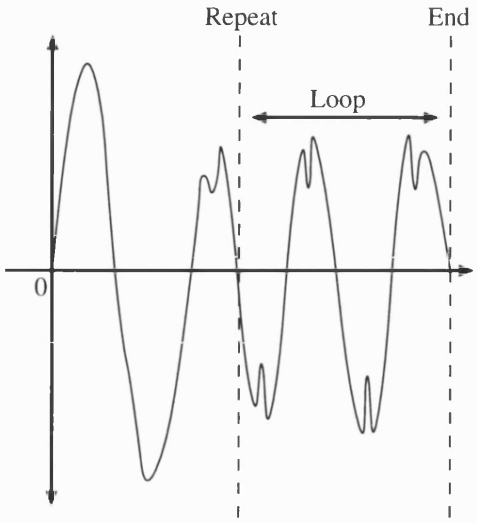

Figure 4-13: *Example of the Timbre Model Used in the Prototypical System*

4.6 System Demonstrations

The following three sections demonstrate the use of the system to produce a number of small example compositions. The first of these is fairly traditional, producing a short choral piece with transformations only being applied to the dimensions of frequency, amplitude, and time. The second example demonstrates an implementation of the discrete dimension of instumental timbre we cited earlier in the text. The final, and most complex, example demonstrates the use of transformations in an implemented stochastic space, or space of probabilistic distributions, to drive a stochastic grain generator.

4.6.1 Frequency, Amplitude and Time

Figure 4-14 shows the structure of a short example composition which was produced using the sound renderer, but without the use of the user interface. Instead the structure was built using EuLisp code which when interpreted created the required objects, rooms, doors, functions, notes etc, and rendered the sound over an interval of $[0, 8]$ seconds. The resulting outputs from progressively building the composition, the steps for which are boxed by dashed lines and numbered in the figure, were each recorded onto audio tape. This tape was then used to demonstrate the method of hierarchical construction, or assembly, in a seminar, given by the author on the 10^{th} March 1995, to members of the Computing Group at the University of Bath. The steps of construction are as follows:

1. Firstly a single room was created into which two note objects pitched at D and A, above middle C, respectively were placed. The amplitudes for the two notes were equal, whilst the timbre for both was given as a sampled voice sound. The sound for this particular room was then rendered and inspected.

2. At this point a room was constructed above the original and functions affecting the frequency parameters were added to the door connecting the two rooms. These functions behave as a composite function, such that any frequency is transposed down one octave, i.e. is halved, and is then further modified by a function which acts as a motif, which is thus dependent upon time. The sound at the new room was rendered, and then listened to.

3. Two new rooms were created below the top room in the hierarchy. Into the first of these new rooms copies of the two notes, found in the first room, were placed. Into the second of the newly created rooms we placed a single note, with the same timbre and amplitude as the others, but pitched at E above middle C. Functions acting as motives were added to the doors connecting these new rooms to the highest room in the hierarchy. The sound at the highest, or root, room was then rendered, producing a sound akin to that of five singing voices in combination.

4. In a similar fashion, a gong sound was added to the composition. Then to demonstrate the control that is made possible by the nodes and arcs model, a glissando was applied to the whole of the choral sound, as if it were simply one sound generating object. The sound for the current state of the example composition was then produced.

5. Finally, to demonstrate that time could be changed, or warped, in the same manner as any other class of parameter, a function was added to make time effectively run four times faster in all parts of the structure below the root room, than was previously the case. The sound for the root room, for the completed composition, was therefore rendered. Rendering was achieved over an interval of time of $[0, 8]$ seconds, at a control rate of 100 Hz, and a sampling rate of 48 kHz. This computation took roughly six minutes, which is obviously far from real-time production. This "slowness" is due to a number of factors, such as the implementation of the renderer being fairly naive. The path rendering is computed in the EuLisp interpreter which frequently has to carry out garbage collection. The track rendering is achieved through compiled C code, but is rather expensive in this case due to the size of the samples used for timbres. The voice sample consists of 19,835 sixteen bit samples, which are stretched, squashed or

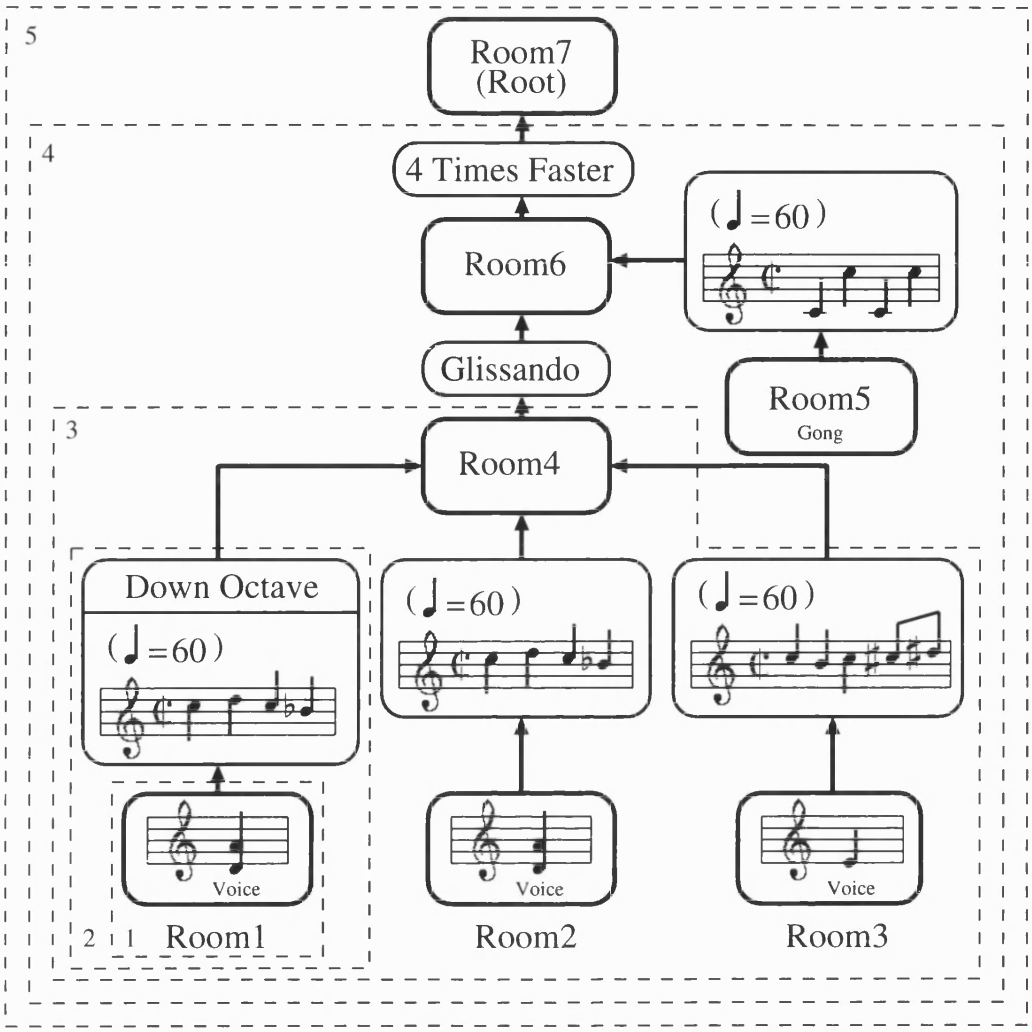

Figure 4-14: *Structure of an Example Composition*

scaled in the final example, one hundred times for every second of sound. The gong sound consists of 42,028 sixteen bit samples, but is not recomputed very often. While the final form for the example composition is fairly complex, there clearly remains scope for improvement in the efficiency of the rendering process.

The primary purpose of the above example was to test the rendering of composite sounds. It does not demonstrate the properties of the system such as being able to experiment at all levels of the composition, or the re-usability of elements that have been experimented upon. These properties can only be demonstrated by the entire system, through the use of the user interface. This has been done on less complex example compositions by creating structures of rooms and doors on the fly, and experimenting

with notes placed in them through the production of functions, generated using the graph editor.

The following two examples detail the more complex dimensions found in the system, i.e. timbral and stochastic dimensions.

4.6.2 Timbre Space

Our dimension of instrumental timbre is defined previously in the text, and is shown in figure 3-6. For the aid of the reader we will, however, restate the definition in the following text. Let $\Gamma = \{violin, viola, cello, contrabasso, flute, oboe, clarinet, bassoon, trumpet, horn, trombone, tuba, soprano, alto, tenor, bass\}$. Let $\Delta = \{string, woodwind, brass, voice\}$. We have the interpretation that $\gamma \in \Gamma \Rightarrow \gamma$ is an instrumental timbre and $\delta \in \Delta \Rightarrow \delta$ is an instrument family. The action of the function Φ in this case is when given a timbre γ, and a family δ, the result is the timbre that is the counterpart of γ in δ, i.e. $\Phi : \Gamma \times \Delta \longrightarrow \Gamma$. That is the instrument in the family that has a similar range. For example $\Phi(viola, voice) = alto$ and $\Phi(flute, string) = violin$. Hence the triple $\Omega = (\Gamma, \Delta, \Phi)$ defines our dimension of instrumental timbre.

Figure 4-15 shows the structure of a short example composition which was produced to demonstrate this particular dimension of instrumental timbre. The steps taken in its construction are described in the following.

1. Firstly a single room was created into which we placed one note object. The timbre for the note is given as *tuba*, the pitch was given as the first D below middle C, whilst the amplitude value is not of importance. The sound thus produced for this room was a single note played with a tuba sound at the pitch given above.

2. Three additional rooms were created. Into each of these a single note object was placed with the pitch and timbre described in figure 4-15. A new room was created above each of these rooms to represent the combination of the sounds from all of them. The sound produced at this room, *Room5*, at this point was, therefore, that of a brass quartet playing a four part chord of D minor. However, functions were then applied to the doors connecting *Room5* to the rooms below it transforming their position in timbre space. Each was set to move by the distance, or route, *woodwind*, shown as *wind* in the diagram, after a certain period of time. For example the tuba part was set to move via *wind* after two seconds, and thus become a bassoon. Hence the effect of the sound produced at *Room5* after these additions was one of a brass quartet transforming into a wind quartet.

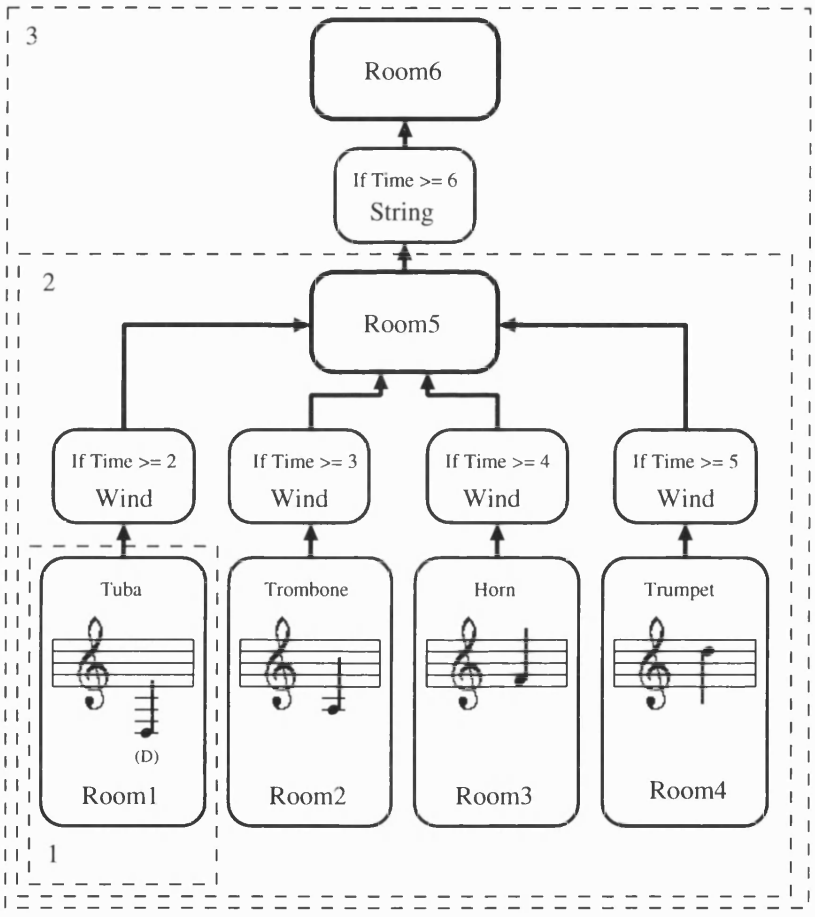

Figure 4-15: *Composition Structure Demonstrating Timbre Space*

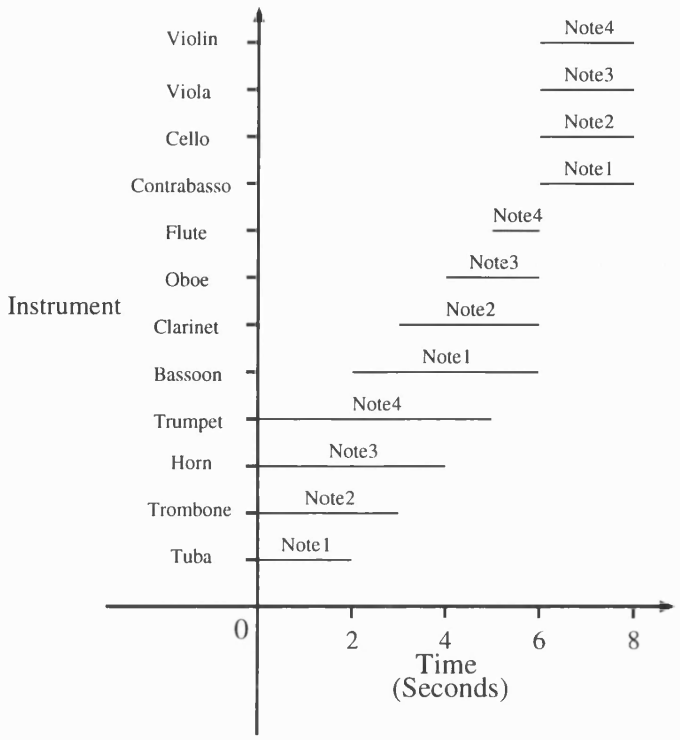

Figure 4-16: *Timbre Paths for Each Note Object*

3. Finally a room was created above *Room5*. A function transforming the position of *Room5*, via the route *string* after six seconds, was applied to the door connecting *Room5* to *Room6*. The eight seconds of sound thus produced at *Room6* was that of the transforming of brass into woodwind as before, but with the addition that after six seconds the wind quartet suddenly changes into a string quartet.

The affect of the transformations on each note, or quartet member, can be seen in figure 4-16 which shows the path of the instrumental timbre parameter for each note object. From this diagram we can follow how the total sound produced evolves over time.

4.6.3 Stochastic Space

Granular synthesis [56] is a technique for specifying complex sounds as the assemblage of a large number of simple sonic quanta, otherwise referred to as grains. Each grain being, effectively, a point in a space of parameters used to drive a pressure function. The specification of a complex sound is therefore achieved via the choice of grains and the specification of the distribution of grains within this space. Figure 4-17 shows a set

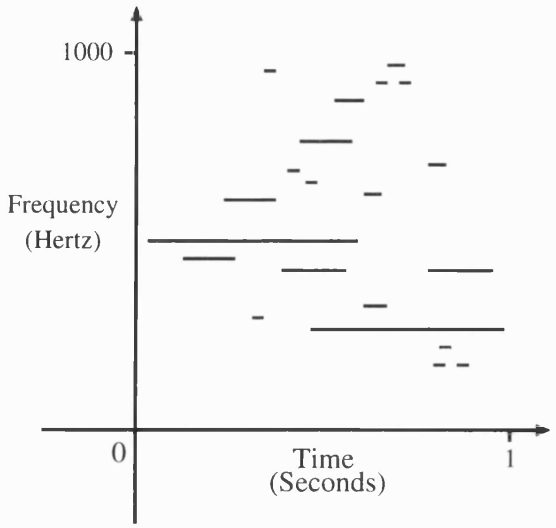

Figure 4-17: *A Distribution of Grains over Frequency and Time*

of grains distributed over frequency and time. Each line in the diagram shows the life of a particular grain over which its amplitude rises from silence to its full value, and returns again to silence.

Due to the large quantity of grains required to produce a sound of reasonable complexity, the distribution of grains is normally achieved via the use of an algorithm. A very common instance of this approach is the use of stochastic processes to determine the distributions of grains, for example the grain clouds of Iannis Xenakis [67].

Granular synthesis is therefore an ideal basis for the testing of stochastic spaces, by which we mean spaces whose points represent probabilistic distributions. A granular synthesis engine, with the following properties, was implemented as part of the system. The engine produces "simple" sinusoidal grains distributed over frequency, amplitude, and time. The engine therefore has parameters governing the probabilistic distribution of the grains over both frequency and amplitude. In addition a real number, greater than zero, governs the average number of grains starting per second, which are uniformly distributed over time, and of a fixed length.

We define our stochastic space, or space of probabilistic distributions, as $\Omega_s = (\Gamma_s, \Delta_s, \Phi_s)$. The set of points in the space is given by $\Gamma_s = S \times \Gamma_p \times \Gamma_p$, where $S = \{uniform, triangular, valley\}$, and Γ_p is the set of points from a dimension $\Omega_p = (\Gamma_p, \Delta_p, \Phi_p)$ over which our parameters values are to be randomly distributed. Therefore given a particular point in this space $\gamma = (d, p_1, p_2) \in \Gamma_s$, d describes one of the possible types of probability density function (pdf), shown in figure 4-19. The values $p_1, p_2 \in \Gamma_p$ describes the range of the pdf, outside of which it is always zero. It

is therefore the case that for each pdf $f : [p_1, p_2] \longrightarrow [0, 1]$ we have $\int_{p_1}^{p_2} f(x)dx = 1$.

Uniform In the uniform distribution the probability of each value in the range $[p_1, p_2]$ is identical. Thus the pdf, for the uniform distribution, $f_u : [p_1, p_2] \longrightarrow [0, 1]$ is given by

$$f_u(x) = \begin{cases} \frac{1}{p_2 - p_1} & \text{if } p_1 \leq x \leq p_2 \\ 0 & \text{otherwise} \end{cases}$$

This constant probability, between these limits, is represented by the value u_h in figure 4-19

Triangular For the distribution referred to as triangular the probability is greatest at the midpoint $m = \frac{p_1+p_2}{2}$, this being given by $h_t = \frac{2}{p_2-p_1}$. Hence, the pdf, for the triangular distribution, $f_t : [p_1, p_2] \longrightarrow [0, 1]$ is given by

$$f_t(x) = \begin{cases} \frac{h_t(x-p_1)}{m-p_1} & \text{if } p_1 \leq x \leq m \\ \frac{h_t(p_2-x)}{p_2-m} & \text{if } m < x \leq p_2 \\ 0 & \text{otherwise} \end{cases}$$

Valley For the distribution shape referred to as valley the probability is greatest at both end points, and zero at the midpoint $m = \frac{p_1+p_2}{2}$. The probability at the end points p_1 and p_2 is given by $h_v = \frac{2}{p_2-p_1}$. Hence, the pdf, for the valley distribution, $f_v : [p_1, p_2] \longrightarrow [0, 1]$ is given by

$$f_v(x) = \begin{cases} \frac{h_v(m-x)}{m-p_1} & \text{if } p_1 \leq x \leq m \\ \frac{h_v(x-m)}{p_2-m} & \text{if } m < x \leq p_2 \\ 0 & \text{otherwise} \end{cases}$$

The set of distances, or routes, for our stochastic dimension is given by $\Delta_s = D \cup M$, where $D = \{uniform, triangular, reflect\}$ and $M = \{1, 2\} \times \Delta_p$. The routes between the points in our dimension can therefore be one of two things. If we have a route $\delta \in D$ then going along this route will lead to a point which may contain a different type of distribution. In order to define the function Φ_s for this dimension will we rename some projection functions by setting $\gamma_s(\gamma) = \Pi_1(\gamma)$, $\gamma_1(\gamma) = \Pi_2(\gamma)$, and $\gamma_2(\gamma) = \Pi_3(\gamma)$ $\forall \gamma \in \Gamma_s$. We therefore define the function $\Phi_s : \Gamma_s \times \Delta_s \longrightarrow \Gamma_s$, evaluated as $\Phi_s(\gamma, \delta)$ where $\delta \in D$, by

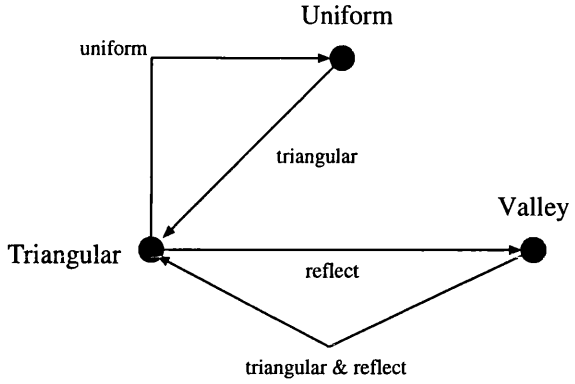

Figure 4-18: *Distribution Shape Changing Routes*

$$\Phi(\gamma,\delta) = \begin{cases} (uniform, \gamma_1(\gamma), \gamma_2(\gamma)) & \text{if } \delta = uniform \\ (triangular, \gamma_1(\gamma), \gamma_2(\gamma)) & \text{if } \delta = triangular \\ (valley, \gamma_1(\gamma), \gamma_2(\gamma)) & \text{if } \delta = reflect \text{ and } \gamma_s(\gamma) = triangular \\ (triangular, \gamma_1(\gamma), \gamma_2(\gamma)) & \text{if } \delta = reflect \text{ and } \gamma_s(\gamma) = valley \\ \gamma & \text{otherwise} \end{cases}$$

Figure 4-18 shows the distribution types and the routes available between them in our stochastic space. For the purpose of clarity only the routes that lead to a different distribution type are shown.

If we have a route $\delta \in M$, then the result of moving via this route is that of arriving at a point where the range of the distribution may have been altered. We therefore define the function $\Phi_s : \Gamma_s \times \Delta_s \longrightarrow \Gamma_s$, evaluated as $\Phi_s(\gamma, \delta)$ where $\delta \in M$, by

$$\Phi(\gamma,\delta) = \begin{cases} (\gamma_s(\gamma), \Phi_p(\gamma_1(\gamma), \delta), \gamma_2(\gamma)) & \text{if } \Pi_1(\delta) = 1 \\ (\gamma_s(\gamma), \gamma_1(\gamma), \Phi_p(\gamma_2(\gamma), \delta))) & \text{if } \Pi_1(\delta) = 2 \end{cases}$$

As an example, which we will use in the demonstration of the stochastic space, consider we have a dimension of frequency in Hertz given by a positive real number, with real number differences, and addition as the means of navigating the routes. That is if our frequency dimension is given by $\Omega_f = (\Re^+, \Re, \Phi_f)$ then for example $\Phi_f(440, -220) = 440 + -220 = 220$. Now let us suppose that in our definition of the stochastic space we have $\Omega_p = \Omega_f$. Then given the point $(uniform, 100, 200) \in \Gamma_s$, and the route $(1, 50) \in M$, we have $\Phi_s((uniform, 100, 200), (1, 50)) = (uniform, 150, 200)$.

Figure 4-20 describes the structure of a short example composition built to demonstrate the use of our stochastic space to control a grain generating object. The steps

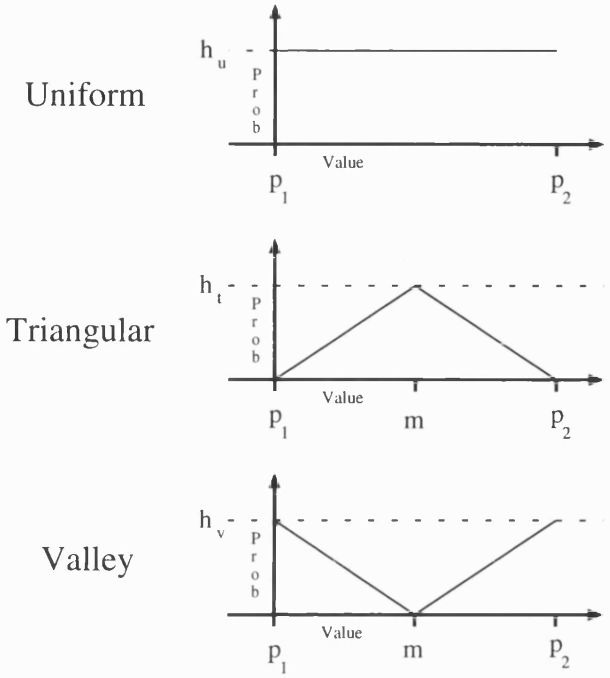

Figure 4-19: *Available Distribution Types*

followed to produce this composition were as follows.

1. A single room was created into which was placed a grain generator. The initial parameters were set such that the generator would produce an average of fifty grains per second. The initial distribution for the pitch of the grains was given as (*uniform*,100,200) i.e frequencies distributed uniformly between 100 Hz and 200 Hz. The amplitude was set to be the same for all grains, and each grain would have the same duration, this being 0.2 seconds. The two seconds of sound produced was thus that of approximately one hundred grains uniformly distributed over this period and the set range of frequencies.

2. A second room was constructed above the first and two functions added to the door between the two rooms. The first of these functions transforms a point in the stochastic space by raising the top limit of the distribution by 800 Hz over a period 0.5 seconds. This position remains fixed for 1 second and then moves back down again over the next 0.5 seconds to reach the point at which it began. The graph of this function can be seen in figure 4-20 in the left hand side of the box above *Room1*. The second transformation function raises the bottom limit of the distribution by 600 Hz over the period of time 0.5 seconds to 1 second. It then returns back to the point at which it began over the next 0.5 seconds and

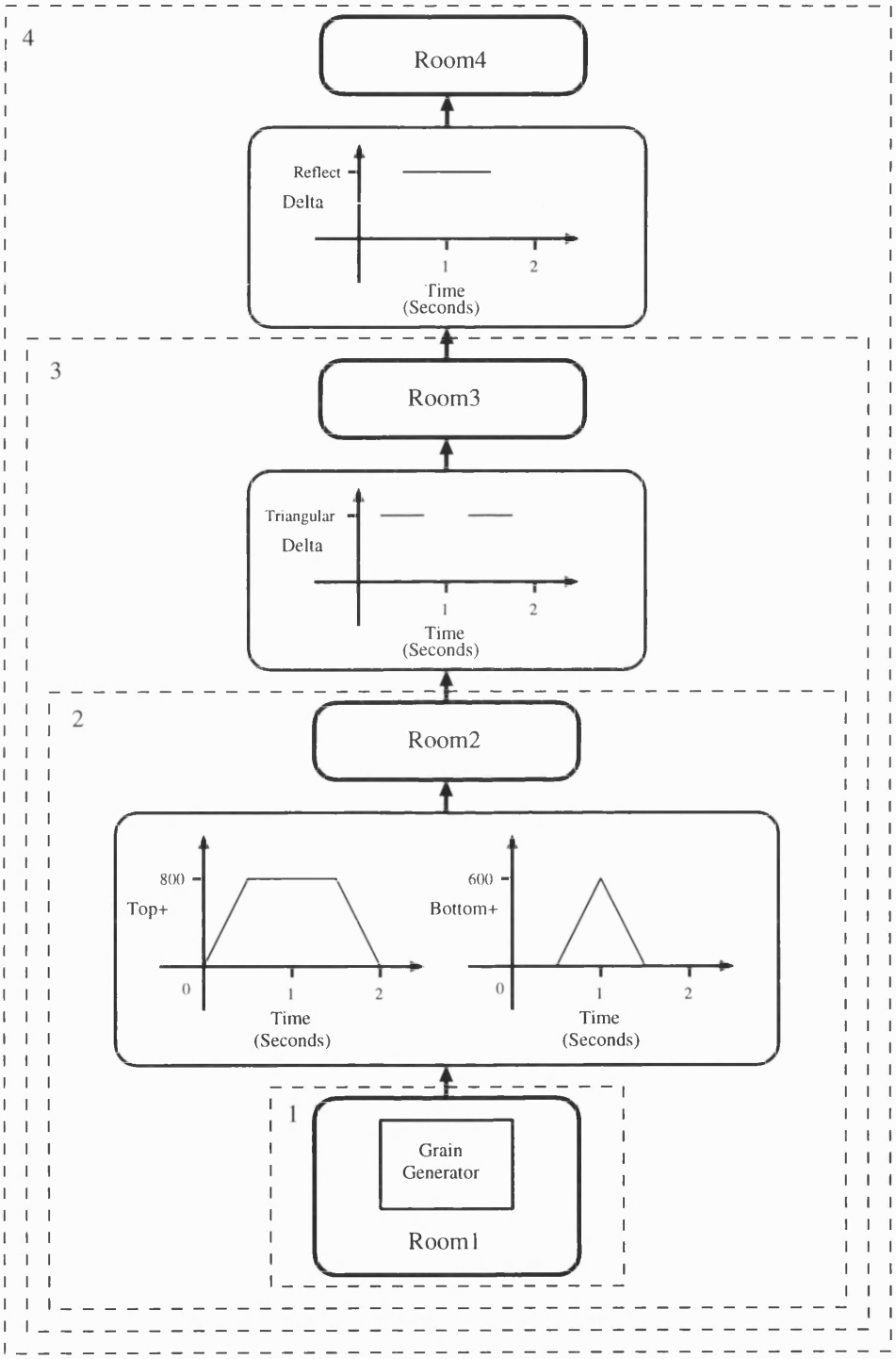

Figure 4-20: *Composition Structure Demonstrating Stochastic Space.*

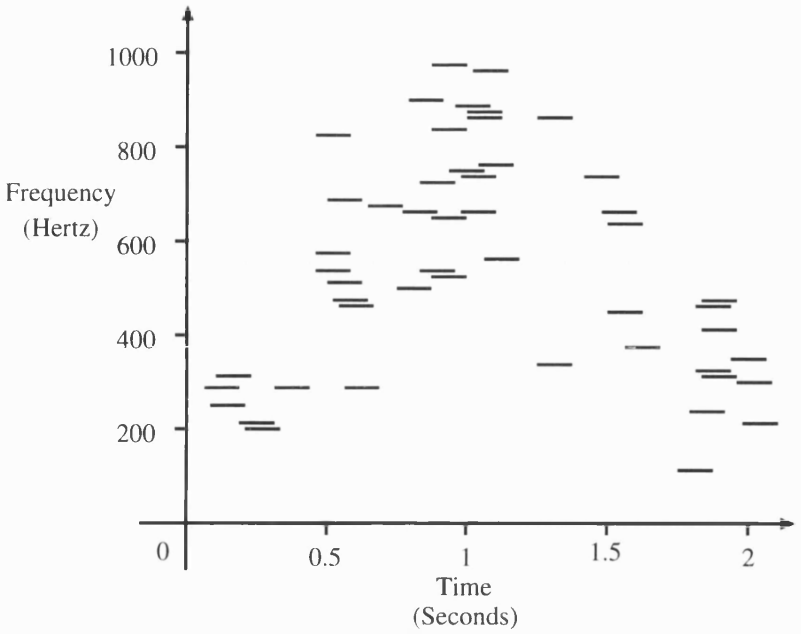

Figure 4-21: *Grains Produced after the First Transformation*

has no affect after this time. The graph of this function can be seen in figure 4-20, in the right hand side of the box above *Room1*. The grains produced due to these two transformations can be seen in figure 4-21[3]. As can be seen from the diagram the trend is for the grains to follow the change in the limits of the uniform distribution governing their pitch.

3. An additional room was constructed above *Room3* and a function added to the newly created door. This function translates the distribution type of a point in the stochastic space. Between the intervals of time 0.25 seconds to 0.75 seconds, and 1.25 seconds to 1.75 seconds, the room below is moved via *triangular*. The graph for this function can be seen in figure 4-20 in the box above *Room2*. The application of this function results in all distribution types becoming *triangular* between these periods. The resulting set of grains produced, and thus sound, was similar[4] to the sound produced without the new transformation between the intervals of time when the transformation did not apply. However, over the intervals of time where the distributions become *triangular* there was a noticeable

[3]For the purpose of clarity only fifty of the grains are shown. In addition all graphs of grains against time were produced via separate experiments without using the main system. This was necessary since rendering the structure within the complete system only produces a sound file as output.

[4]Not necessarily the same since only the distributions remain unaltered. The grains are still randomly distributed according to the distributions. Hence the set of grains produced is not likely to be the same.

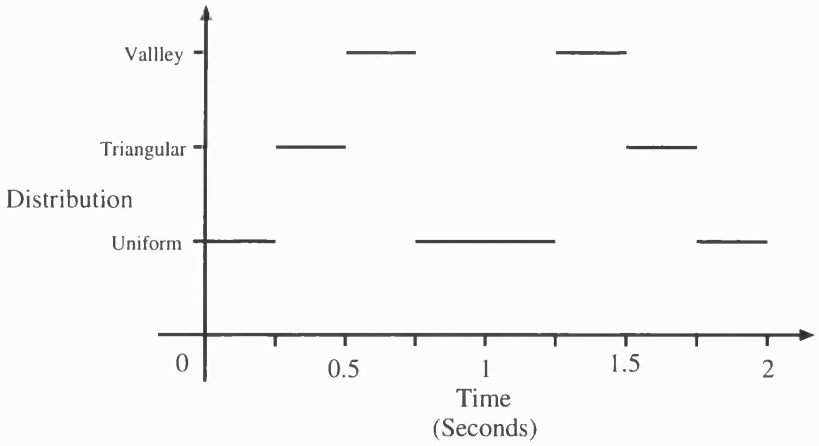

Figure 4-22: *Path of the Distribution Type after the Final Transformation*

concentration of grains around the midpoint of the range of the distribution.

4. Finally a room was constructed above *Room4* and a function added to the new door. This function also translates the distribution part of a point in the stochastic space. However the resulting motion is via the distance, or route, *reflect*, and acts over the interval of time 0.5 seconds to 1.5 seconds. This results in any *triangular* distributions becoming *valley* distributions, and vice versa. The graph for this function can be seen in figure 4-20 in the box above *Room3*. The affect of all these transformations in total on the path of the stochastic distribution, part of the frequency distribution parameter in the grain generator, can be seen in figure 4-19. This path, and the known path of the distribution limits, accords with the set of grains produced and is shown in figure 4-23. The overall sound effect was therefore one of a cloud of sounds rising and then falling in pitch over a period of two seconds, with the clouds being shaped according to the controlling distribution at a particular moment.

The production of this set of grains, via the motion of the grain generator through our stochastic space, concludes our section on system demonstrations.

4.7 Remarks on the Prototypical System

The purpose of the implementation of the prototypical system was to provide a platform on which the nodes and arcs model could be tested, to see if in practice it can be cited as a solution to the problem of score specification, as well as in theory. The implementation therefore is only concerned with the properties of the system that are new. This was

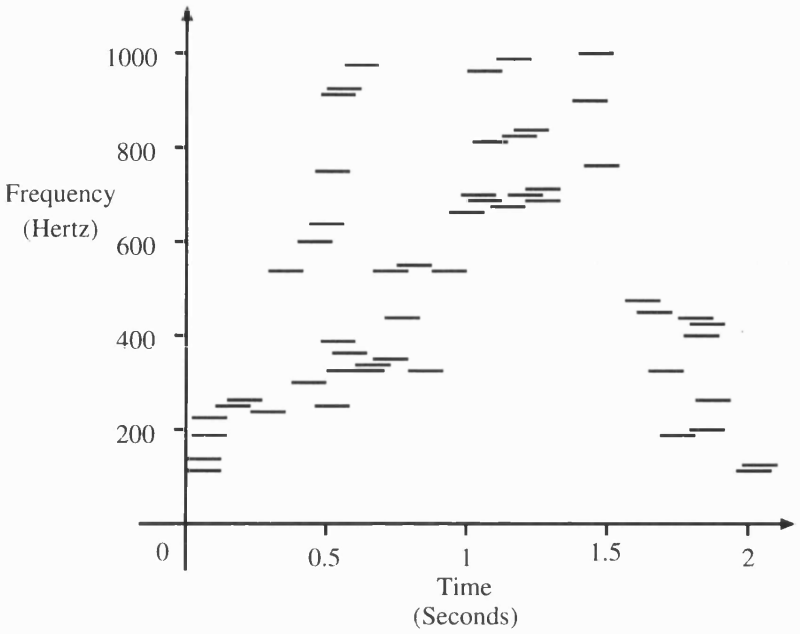

Figure 4-23: *Grains Produced after the Final Transformation.*

achieved in the current implementation, and through tests on the sound renderer, and the complete system, the nodes and arcs model was shown to provide the solution that was desired.

Part III

Evaluation : Summary, Future Research, and Conclusions

Chapter 5

Evaluation

5.1 Summary

In the preceding text we have traced the problems relating to computer sound synthesis back to their source. This was found to be a property of the digital representation of sound itself. Although the digital representation of sound is completely general, the amount of data required for this generality is too large to be specified manually. This has lead to sound being specified using various synthesis techniques, which require a smaller amount of data to drive them, but produce a loss in generality. This problem we referred to as the fundamental problem of digital sound synthesis.

We further demonstrated that this problem had been split into the two component problems of instrument specification and score specification. Both of these component problems was shown to inherit a problem relating to generality from their parent problem. In the case of instrument specification it was shown that for any particular synthesis technique there will exist types of wave forms and timbres for which it is not general enough. For example FM synthesis is not general enough to produce high quality voice synthesis. If it is the case that a particular synthesis technique is general enough for all types of timbres, then there will exist wave forms and timbres for which it is too general. For example a general physical model would be rather an over elaborate method of producing a wave that could easily be synthesised using FM. The most general method would of course be the specification of samples, which can be said to be too general, by reference to the fundamental problem. The unit generator model, however, was shown to provide a solution to the generality problem of instrument specification, provided that the set of unit generators is extensible.

In score specification the problem was uncovered by the fact that for any particular score writing system, or method, there will exist musical ideas for which it is not general

enough to notate, or there will exist musical ideas for which it is too general, in the sense that to use the notation in this instance would be a tedious exercise. As examples we noted that common western notation is perfectly adequate for notating a piano sonata, say, but is not general enough to notate a microtonal composition. A graph notation is completely general, but would be rather too general to make the notation of a piano sonata anything other than tedious and inefficient. This becomes a serious problem in the situation in which we wish to construct a composition, or section of a composition, from parts for which the most appropriate method of notation is not appropriate for the other parts. We therefore argued that the best way to solve this difficulty was to specify each part using the most appropriate notation system, and then combine the results to form the piece we desire. Under the paradigm of experimental computer music (ECM), which we showed to be desirable, the results of the combination of parts, which we called composite sounds, must be subject to the same type of experimentation as the pieces from which they are built. We further showed that experimentation on a composite sound should not affect the parts from which it is built. On examination of current systems for sound synthesis we found this property to be lacking in all of them.

As a solution to this unsolved problem, a new model was described for the construction of composite sounds. The nodes and arcs model, which views the construction as taking place in what we have called *parameter-space*, was shown to overcome this problem. This was achieved due to the fact that the modification of sound, which is the result of experimentation, is achieved via motion through the parameter-space, and not by altering the internal structure of any sound generating object. The theory behind these structures of nodes and arcs was described, and the method for rendering the sound from them was given.

With the use of this new model, and the theory behind it, a prototypical system was described. The implementation of this system, which has been realised, provided a means for testing the aspects of a complete system, that were new, i.e. the aspects that were a consequence of the nodes and arcs model. The resulting system was shown to have been tested on small example compositions, and demonstrated, to establish that the nodes and arcs model, and hence systems which implement it, provides a solution to the problem of score specification under the paradigm of experimental computer music.

5.2 Areas for Future Research

5.2.1 Exploring Dimensions

Since the control, and modification, of sound in the nodes and arcs model is achieved via motion through a number of dimensions, the combination of which we called a parameter-space, the power of the model may be increased by the implementation of more complex dimensions than simply frequency or amplitude. One such complex dimension is that of instrumental timbre. This has been achieved to an extent via the implementation of a discrete dimension of instrumental timbre. More impressive effects could be achieved, however, via the implementation of a continuous dimension of timbre. The continuous motion of a composite sound in such a dimension could yield simple control over effects such as the gradual, i.e continuous transformation of a string quartet into a wind quartet, or four part choir, over time, as opposed to the discrete alterations in our example. In general the problem of sound morphing or mutation [48], could be explored by examining which paths from one point, or parameterised sound, to another yield the best aesthetic results.

The control of sound generators which produce algorithmic compositions, specified by a fixed set of parameters, is another possible application. In the case of stochastic music we have demonstrated the affect of motion in a dimension of probabilistic distributions on a grain generator. There is still of course room for greater generalisation and control in the realm of stochastic space. With the implementation of more general dimensions, and additional sound generators dependent upon them, the increased power in sound control, provided by the implementation of such dimensions, could be enormous.

5.2.2 Implementing a Complete System

The implementation of a greater number of score editing tools, more complex sound generating objects, and a faster rendering process etc, are all necessary for the production of a complete system for experimental computer music. Much of this work is, however, simply a matter of coding and as such is not strictly part of research, as it does not add to the theoretical strength of the model, although it may demonstrate it more thoroughly. The analysis of what score writing tools are necessary to adequately span the set of musical ideas is of interest. The results of this analysis could be used to construct a sort of set of "unit generators for scores." The author suspects that, like unit generators used in the realm of instrument specification, the set must be extensible, probably with the use a score writing system designer, or model, such as the three dimensional transparent glyph (3TG) model of Glendon Diener [7].

5.2.3 User Interface Issues

An important aspect to consider in the construction of a complete system is the design of the user interface. This entails the consideration of what is a viable paradigm to base the interface upon. In the prototypical system the nodes and arcs model was viewed as a set of rooms connected by doors, with all system objects inhabiting the rooms. As was mentioned at the time, there may exist other viable paradigms for the viewing of the model. Perhaps, for example, we could view the hierarchy as boxes within boxes with the system's objects found inside these boxes. It is therefore obvious that an investigation into these possibilities is a useful area for future research. It is suspected that the result of these investigations may be the realisation that the most appropriate paradigm is one that may be altered, or completely specified, by the user of the system, should he/she wish to do so.

One property of the nodes and arcs model, and hence the prototypical system, that has been deliberately emphasised is its capacity for enabling the user to work at any particular level of the composition at any time. The emphasising of this point has resulted in the exclusion of the usefulness of viewing, and manipulating, the compositional structure as a whole, in all its complexity. One particular way of doing this may be to view the composition as a kind of skeleton which moves in a high dimensional parameter space. It would therefore be useful to investigate methods for the graphical manipulation of such structures. This is likely to entail an examination of similar techniques found in computer graphics, and in particular computer animation. However, the addition of such elements must lead to a more complete, varied, and thus more effective user interface for the system.

5.2.4 Pedagogical Uses

One of the remarks made concerning the paradigm of experimental computer music was that it was in some way similar to play. This correspondence is made clearer by the nodes and arcs model which allows for the experimentation with sound and music in a similar way to the use of a child's building kit, which we described earlier. This could be used as the basis for a computer music system, built upon the nodes and arcs model, and with suitable tools and objects, for music composition via experimentation and play, aimed at non-musicians. This, as Iannis Xenakis has stated, would be one way in which to get people outside of the music community to take an active role in music production.

5.3 Conclusions

This thesis has outlined what is required of a complete and general system for experimental computer music. The nodes and arcs model provides the first solution to the problem of score specification under the paradigm of experimental computer music. This solution has been tested and found to be valid via the implementation and examination of a prototypical system based on the new nodes and arcs model. The prototypical system is in no way a complete and general system for computer sound synthesis. It does, however, serve to illustrate the theoretical strength of the nodes and arcs model in a practical setting. Given the solution to the score specification problem supplied by the nodes and arc model, and the existence of systems such as DMIX, which implement solutions to the other cited problems of computer sound synthesis, we can conclude that a complete system for general computer sound synthesis is implementable, using the new model as its base. This potential, but as yet imaginary, system would appear to sidestep the fundamental problem of digital sound synthesis, by providing a means for a user to deal with all aspects of sound and music specification at the level of generality that is appropriate.

Bibliography

[1] *MIDI specification 1.0.* International MIDI association, 1983.

[2] R. Bidlack. Chaotic systems as simple (but complex) compositional algorithms. *Computer Music Journal*, 16(3), 1992.

[3] Bertrand Couasnon and Bernard Retif. Using a grammar for a reliable full score recognition system. In *ICMC Proceedings*, 1995.

[4] Shawn L. Decker, Gary S. Kendall, Brian L. Schmidt, M. Derek Ludwig, and Daniel J. Freed. A modular environment for sound synthesis and composition. *Computer Music Journal*, 10(4), 1986.

[5] Peter Desain and Henkjan Honing. Tempo curves considered harmful : A critical review of the representation of timing in computer music. In *ICMC Proceedings*, 1991.

[6] Glendon R. Diener. TTrees : A tool for the compositional environment. *Computer Music Journal*, 13(2), 1989.

[7] Glendon R. Diener. *Modeling Music Notation : A Three-Dimensional Approach.* PhD thesis, Department of Music, Stanford University, San Francisco CA, U.S.A, 1990.

[8] Glendon R. Diener. A visual programming environment for music notation. In *ICMC Proceedings*, 1992.

[9] Richard Dobson and John Fitch. Experiments with chaotic oscillators. In *ICMC Proceedings*, 1995.

[10] Charles Dodge. A musical fractal. *Computer Music Journal*, 12(3), 1988.

[11] Mark M. Dolson. The phase vocoder : A tutorial. *Computer Music Journal*, 10(4), 1986.

[12] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics Principles and Practice.* Addison Wesley, second edition, 1990.

[13] Ichiro Fujinaga, Bo Alphonce, Bruce Pennycook, and Glendon Diener. Interactive optical music recognition. In *ICMC Proceedings*, 1992.

[14] Dan Gang and Daniel Lehmann. An artificial neural net for harmonizing melodies. In *ICMC Proceedings*, 1995.

[15] Michael Gogins. Gabor synthesis of recurrent iterated function systems. In *ICMC Proceedings*, 1995.

[16] Isao Hidaka, Masataka Goto, and Yoichi Muraoka. An automated jazz accompaniment system reacting to solo. In *ICMC Proceedings*, 1995.

[17] Frode Holm. Understanding FM implementations : A call for common standards. *Computer Music Journal*, 16(1), 1992.

[18] J. N. Holmes. *Speech Synthesis and Rocognition.* Van Nostrand Reinhold, 1988.

[19] Andrew Horner and Lydia Ayers. Harmonization of musical progressions with genetic algorithms. In *ICMC Proceedings*, 1995.

[20] Andrew Horner and David E. Goldberg. Genetic algorithms and computer-assisted music composition. In *ICMC Proceedings*, 1991.

[21] Julius Orion Smith III. Viewpoints on the history of digital synthesis. In *ICMC Proceedings*, 1991.

[22] William F. Punch III. An algorithmic approach to composition based on dynamic hierarchical assembly. In *ICMC Proceedings*, 1991.

[23] Bruce L. Jacob. Composing with genetic algorithms. In *ICMC Proceedings*, 1995.

[24] Gary S. Kendall and Lonny Chu. The sound of fruits and vegetables: Scientific visualization with auditory tokens. In *ICMC Proceedings*, 1995.

[25] Gregory Kramer and Stephen Ellison. AUDIFICATION: The use of sound to display multivariate data. In *ICMC Proceedings*, 1991.

[26] Paul Lansky. *CMix Release notes and Manuals.* Department of Music, Princeton University, 1990.

[27] Jeremy Leach and John Fitch. The application of differential equations to the modelling of musical change. In *ICMC Proceedings*, 1995.

[28] Jeremy Leach and John Fitch. Nature, music, and algorithmic composition. *Computer Music Journal*, 19(2), 1995.

[29] Harry B. Lincoln. *The Computer and Music.* Cornell University Press, 1970.

[30] Henning Lohner. Interview with Iannis Xenakis. *Computer Music Journal*, 10(4), 1986.

[31] Gerard Marino, Jean-Michel Raczinski, and Marie-Helene Serra. A description of the UPIC system. In *ICMC and Festival at Delphi*, 1992.

[32] Patricia McLendon. *Graphics Library Programming Guide.* Silicon Graphics, Inc, 1991.

[33] Keith McMillan, David L. Wessel, and Matthew Wright. The ZIPI music parameter description language. *Computer Music Journal*, 18(4), 1994.

[34] Norris D. McWhirter. *The Guiness book of records.* Guiness, 1982.

[35] David K. Mellinger and Bernard M. Mont-Reynaud. SoundExplorer: A workbench for investigating source seperation. In *ICMC Proceedings*, 1991.

[36] F. J. Moore. *Elements of Computer Music.* Prentice Hall, 1990.

[37] Joseph Derek Morrison and Jean-Marie Adrien. MOSAIC: A framework for modal synthesis. *Computer Music Journal*, 17(1), 1993.

[38] Ken'ichi Ohya. A sound synthesis by recurrent neural networks. In *ICMC Proceedings*, 1995.

[39] Daniel V. Oppenheim. SHADOW: An object-oriented performance system for the DMIX environment. In *ICMC Proceedings*, 1991.

[40] Daniel V. Oppenheim. Towards a better software-design for supporting creative musical activity. In *ICMC Proceedings*, 1991.

[41] Daniel V. Oppenheim. Compositional tools for adding expression to compositions in DMIX. In *ICMC Proceedings*, 1992.

[42] Daniel V. Oppenheim. DMIX - a multi faceted environment for composing and performing computer music: its design, philosophy, and implementation. In *ICMC and Festival at Delphi*, 1992.

[43] Daniel V. Oppenhiem. Slappability: A new metaphor for human computer interaction. In *ICMC Proceedings*, 1993.

[44] Mark H. Overmars. *Form Library : A graphical user interface toolkit for Silicon Graphics workstations : version 2.3.* Available via anonymous ftp at ftp.cs.ruu.nl, 1995.

[45] Julian Padget and Greg Nuyens. *The EuLisp language definition.* Available via anonymous ftp at midge.bath.ac.uk.

[46] Mark Pearson and David M. Howard. A musicians approach to physical modelling. In *ICMC Proceedings*, 1995.

[47] Walter Piston. *Orchestration.* Gollancz, 1955.

[48] Larry Polansky and Tom Erbe. Spectral mutation in soundhack : A brief description. In *ICMC Proceedings*, 1995.

[49] Richard Polfreman and John Sapsford-Francis. A human factors approach to computer music systems user-interface design. In *ICMC Proceedings*, 1995.

[50] Stephen Travis Pope. Three packages for software sound synthesis. *Computer Music Journal*, 17(2), 1993.

[51] Stephen Travis Pope. Computer music workstations I have known and loved. In *ICMC Proceedings*, 1995.

[52] Miller Puckett. Is there life after MIDI ? In *ICMC Proceedings*, 1994.

[53] Jean-Michel Raczinski, Gerard Marino, and Marie-Helene Serra. New UPIC system demonstration. In *ICMC Proceedings*, 1992.

[54] Robert Reimann, Claudia Lohnes, and Kevin Walsh. *Graphics Library Windowing and Font Library Programming Guide with Font Library Man Pages.* Silicon Graphics, Inc, 1991.

[55] Dominique M. Richard. Why simulate when we can get the real thing ? In *ICMC Proceedings*, 1991.

[56] Curtis Roads. Introduction to granular synthesis. *Computer Music Journal*, 12(2), 1988.

[57] Axel Robel. Neural networks for modeling time series of musical instruments. In *ICMC Proceedings*, 1995.

[58] R. Rowe, B. Garton, P. Desain, H. Honing, R. Dannenburg, D. Jacobs, S. Pope, M. Puckett, D. Jacobs, C. Lippe, Z. Settel, and G. Lewis. Putting Max in perspective. *Computer Music Journal*, 17(2), 1993.

[59] Gregory J. Sandell. A library of orchestral instrument spectra. In *ICMC Proceedings*, 1991.

[60] Johan Sundberg. *The Science of Musical Sounds.* Academic Press, Inc, 1991.

[61] H. Taube. Common music : A music composition language in common lisp and clos. *Computer Music Journal*, 15(2), 1991.

[62] C P Tsang and M Aiken. Harmonizing music as a discipline of constraint logic programming. In *ICMC Proceedings*, 1991.

[63] Guido van Rossum. *Audio File Formats (FAQ) version 2.10.* Available via anonymous ftp at ftp.cse.cz, 1993.

[64] Barry Vercoe. *CSound Manual and Release Notes.* Available via anonymous ftp at media-lab.mit.edu, 1991.

[65] Ammon Wolman, James Choi, Shahad Asgarzadeh, and Jason Kahana. Recognition of handwritten music notation. In *ICMC Proceedings*, 1992.

[66] Matthew Wright. Answers to frequently asked questions about ZIPI. *Computer Music Journal*, 18(4), 1994.

[67] Iannis Xenakis. *Formalized Music.* Bloomington: Indiana University Press, 1971.

[68] Iannis Xenakis. More thorough stochastic music. In *ICMC Proceedings*, 1991.